Оглавление

[JDBC, ORM, Hibernate, JPA 7](#_Toc44613878)

[JDBC (Java Database Connectivity) 7](#_Toc44613879)

[ORM 7](#_Toc44613880)

[Hibernate 7](#_Toc44613881)

[JPA (Java Persistency API) 7](#_Toc44613882)

[JDBC и Hibernate 7](#_Toc44613883)

[JPA и JDBC 10](#_Toc44613884)

[Что такое хибер 10](#_Toc44613885)

[Важные интерфейсы Hibernate: 11](#_Toc44613886)

[Основные интерфейсы JPA 12](#_Toc44613887)

[EntityManager и Session 13](#_Toc44613888)

[EntityManagerFactory и SessionFactory 13](#_Toc44613889)

[Session и EntityManager 14](#_Toc44613890)

[Когда вызываешь метод createQuery() какой интерфейс получаешь 14](#_Toc44613891)

[orphanRemoval 14](#_Toc44613892)

[Hibernate proxy (lazy load) 14](#_Toc44613893)

[Транзакции в Hibernate 15](#_Toc44613894)

[В каком случае я могу восстановить удаленную сущность 16](#_Toc44613895)

[Что такое PersistenceContext (это и есть кэш 1-го уровня у EntityManager) 17](#_Toc44613896)

[Как отсоединить сущность от контекста, какими методами 20](#_Toc44613897)

[Как можно конфигурировать хибернейт — какими способами (SessionFactory & EntityManagerFactory) 21](#_Toc44613898)

[На какой стороне инкрементится id — на стороне базы или хибернейта 23](#_Toc44613899)

[Что означает двусторонняя связь. Как это отобразится в коде 23](#_Toc44613900)

[Что произойдет если не поставить mappedBy на @OneToOne @OneToMany — почему выгодно ставить mapped by на этих типах связей 25](#_Toc44613901)

[Университеты и Студенты — @MаnyToMany — можно ли обойтись просто этой аннотацией без @JoinTable и @JoinColumn 26](#_Toc44613902)

[При аннотации @ManyToMany — ОБА ВЛАДЕЛЬЦЫ СВЯЗИ 26](#_Toc44613903)

[@ElementCollection 27](#_Toc44613904)

[OrphanRemoval 27](#_Toc44613905)

[Почему есть Cascade Removal и orphanRemoval — есть еще один случай — когда есть разница? А при удалении разницы нет — в каком случае есть? 28](#_Toc44613906)

[Метод unWrap() 29](#_Toc44613907)

[Каким методом очищается кэш 1-го уровня 30](#_Toc44613908)

[EntityManager 30](#_Toc44613909)

[Основные аннотации 32](#_Toc44613910)

[@Entity 32](#_Toc44613911)

[Требования к Entity в JPA. 32](#_Toc44613912)

[@Table 33](#_Toc44613913)

[@Access 33](#_Toc44613914)

[@Id (первичный ключ) 34](#_Toc44613915)

[Первичный ключ 34](#_Toc44613916)

[@GenerationType (стратегии генерации) 35](#_Toc44613917)

[@EmbeddedId 36](#_Toc44613918)

[@Basic 36](#_Toc44613919)

[@Column 37](#_Toc44613920)

[@Basic и @Column 37](#_Toc44613921)

[@GeneratedValue 37](#_Toc44613922)

[@Cascade 38](#_Toc44613923)

[@PrimaryKeyJoinColumn 38](#_Toc44613924)

[Встраиваемый класс (@Embedded) 38](#_Toc44613925)

[@MappedSuperClass 39](#_Toc44613926)

[Три стратегии построения иерархии 39](#_Toc44613927)

[Маппинг enum 40](#_Toc44613928)

[Маппинг дат 40](#_Toc44613929)

[!!!Как смапить коллекцию примитивов 40](#_Toc44613930)

[Типы связей 40](#_Toc44613931)

[Владелец связи (mappedBy). 41](#_Toc44613932)

[Каскадирование 41](#_Toc44613933)

[FetchType 43](#_Toc44613934)

[Жизненный цикл сущности 44](#_Toc44613935)

[new 44](#_Toc44613936)

[managed 44](#_Toc44613937)

[detached 44](#_Toc44613938)

[removed 44](#_Toc44613939)

[Операции над сущностью в разных состояниях жизненного цикла 44](#_Toc44613940)

[persist() 44](#_Toc44613941)

[remove() 44](#_Toc44613942)

[merge() 45](#_Toc44613943)

[refresh() 45](#_Toc44613944)

[detach() 46](#_Toc44613945)

[Кеширование (уровни кэширования, @Cacheable, @Cache, ehcache). 46](#_Toc44613946)

[@JoinColumn 48](#_Toc44613947)

[@JoinTable 49](#_Toc44613948)

[@OrderBy 50](#_Toc44613949)

[@OrderColumn — как работает, где ставится 51](#_Toc44613950)

[Различия между @OrderBy и @OrderColumn — пример с БД 51](#_Toc44613951)

[!!! Transient 51](#_Toc44613952)

[Блокировки (оптимистические и пессимистические). 51](#_Toc44613953)

[Как работает первый уровень кэша: когда он есть, когда нет, к чему он привязан (к какому объекту) 53](#_Toc44613954)

[К какому объекту привязан кэш второго уровня (к EntityManagerFactory) 57](#_Toc44613955)

[Как настроить кэш второго уровня. 58](#_Toc44613956)

[Какой кэш еще есть. Кэш запросов — как настроить. Желательно понимать как объекты хранятся в кэше второго уровня и в кэше запросов. 60](#_Toc44613957)

[Как контролировать объекты второго уровня кэша: как удалить, как посмотреть. 60](#_Toc44613958)

[HQL, JPQL и SQL 61](#_Toc44613959)

[Criteria API 61](#_Toc44613960)

[n + 1 select (описание и решения) 62](#_Toc44613961)

[Entity Grpah 65](#_Toc44613962)

### JPA и JDBC

Основное различие — уровень абстракции:

JDBC — стандарт низкого уровня для взаимодействия с БД посредством SQL.

JPA (Java Persistence API) — спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных БД в удобном виде. Сама Java не содержит реализации JPA, однако существует много реализаций данной спецификации от разных компаний (открытых и нет). Это не единственный способ сохранения java объектов в БД (ORM систем), но один из самых популярных в Java мире.

### Что такое хибер

Hibernate одна из самых популярных открытых реализаций спецификации JPA. JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA).

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения (object-relational mapping — ORM). Она представляет собой свободное программное обеспечение с открытым исходным кодом (open source). Данная библиотека предоставляет легкий в использовании каркас (фреймворк) для отображения объектно-ориентированной модели данных в традиционные реляционные БД.

### Основные интерфейсы JPA

∙ EntityManagerFactory — фабричный класс EntityManager. Он создает и управляет несколькими экземплярами EntityManager.

∙ EntityManager — интерфейс; он управляет операциями сохранения на объектах. Это работает как фабрика для экземпляра Query.

∙ Entity-сущности — постоянные объекты, хранящиеся в виде записей в БД.

∙ EntityTransaction — имеет непосредственное отношение к EntityManager. Для каждого EntityManager операции поддерживаются классом EntityTransaction.

∙ Persistence — этот класс содержит статические методы для получения EntityManagerFactory экземпляра.

∙ Query — этот интерфейс реализуется каждым поставщиком JPA для получения реляционных объектов, соответствующих критериям.

### EntityManager и Session

Реализация JPA EntityManager является оберткой реализации Hibernate Session. Session расширяет интерфейс EntityManager. Если вы выбираете путь JPA, то всегда имеете возможность быстро перейти, на другие реализации JPA — EclipseLink, OpenJPA, DataNucleus. При желании, можно работая с EntityManager дотянуться через unwrap до интерфейса Session и воспользоваться его функционалом.

### EntityManagerFactory и SessionFactory

Hibernate SessionFactory расширяет JPA EntityManagerFactory.

Таким образом, Hibernate SessionFactory это также JPA EntityManagerFactory.

Оба SessionFactory и EntityManagerFactory содержат метаданные отображения сущностей и позволяют вам создавать Hibernate Session или EntityManager.

### Session и EntityManager

Так же как SessionFactory и EntityManagerFactory, Hibernate Session расширяет JPA EntityManager. Итак, все методы, определенные в EntityManager, доступны в Hibernate Session.

### Когда вызываешь метод createQuery() какой интерфейс получаешь

При вызове createQuery() получаем интерфейс Query.

Это объектно-ориентированное представление запроса Hibernate. Запрос экземпляра Query получается вызовом Session.createQuery().

### 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 ещё 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().

### В каком случае я могу восстановить удаленную сущность

Refresh

∙ Если статус Entity managed, то в результате операции Refresh будут восстановлены все изменения из БД данного Entity, также произойдет refresh всех каскадно зависимых объектов.

### Что такое PersistenceContext (это и есть кэш 1-го уровня у EntityManager)

PersistenceContext (контекст постоянства) находится между клиентским кодом и БД. Это промежуточная область, где постоянные данные преобразуются в сущности, готовые для чтения и изменения клиентским кодом.

Он отслеживает все загруженные данные, отслеживает изменения этих данных и несет ответственность за то, чтобы в конечном итоге синхронизировать любые изменения обратно в БД в конце транзакции.

Hibernate, использует контекст постоянства для управления жизненным циклом объекта в приложении.

Экземпляр EntityManager связан с PersistenceContext. PersistenceContext — это набор экземпляров объекта, в котором для любого постоянного идентификатора объекта существует уникальный экземпляр объекта. В PersistenceContext экземпляры сущности и их жизненный цикл управляются. EntityManager API используется для создания и удаления постоянных экземпляров сущностей, для поиска сущностей по их первичному ключу и для запросов к сущностям.

PersistenceContext — это кэш первого уровня, в котором все объекты извлекаются из БД или сохраняются в БД. PersistenceContext отслеживает любые изменения, внесенные в управляемый объект. Если что‑то меняется во время транзакции, то объект помечается как грязный. Когда транзакция завершается, эти изменения сбрасываются в постоянное хранилище.

EntityManager — это интерфейс, который позволяет нам взаимодействовать с контекстом постоянства. Всякий раз, когда мы используем EntityManager, мы фактически взаимодействуем с контекстом постоянства.

PersistenceContext доступны в двух типах:

∙ в области транзакций;

∙ расширенной области.

Транзакционный контекст персистентности.

Контекст постоянства транзакции привязан к транзакции. Как только транзакция заканчивается, объекты, присутствующие в контексте постоянства, будут сброшены в постоянное хранилище.

Когда мы выполняем какую-либо операцию внутри транзакции, EntityManager проверяет PersistenceContext. Если он существует, он будет использован. В противном случае это создаст PersistenceContext.

Тип контекста персистентности по умолчанию — PersistenceContextType.TRANSACTION. Чтобы указать EntityManager использовать контекст постоянства транзакции, мы просто аннотируем его с помощью @PersistenceContext:

|  |  |
| --- | --- |
| 1.  2. | @PersistenceContext  private EntityManager entityManager; |

Расширенный персистентный контекст

Расширенный контекст постоянства может охватывать несколько транзакций. Мы можем сохранить сущность без транзакции, но не можем сбросить ее без транзакции.

Чтобы сказать EntityManager использовать контекст персистентности расширенной области, нам нужно применить атрибут type @PersistenceContext:

|  |  |
| --- | --- |
| 1.  2.  3. | @PersistenceContext(  type = PersistenceContextType.EXTENDED)  private EntityManager entityManager; |

### Как отсоединить сущность от контекста, какими методами

Когда сущность только создана и записана в БД или когда наоборот, прочитана из БД, она входит в PersistenceContext и обладает неким экземпляром Session, который ей управляет. Однако из этого состояния она может внезапно перейти в состояние «отделенная» (detached). В этом состоянии сущность не связана со своим контекстом (отделена от него) и нет экземпляра Session, который бы ей управлял.

Перейти в это состояние сущность может по следующим причинам:

∙ Явный перевод из persisted в detached вызовом метода evict() у Session.

∙ Сброс контекста методом clear() у Session.

∙ Явное закрытие сессии методом close().

∙ Неявное закрытие сессии связанное с удалением объекта Session.

Над detached объектом нельзя выполнять операции, которые требуют наличия PersistenceContext.

detached сущность можно вернуть в состояние persisted вызовами merge(), lock() или update(), но не saveOrUpdate().

### Как можно конфигурировать хибернейт — какими способами (SessionFactory & EntityManagerFactory)

Существует четыре способа конфигурации работы с Hibernate :

∙ используя аннотации;

∙ hibernate.cfg.xml;

∙ hibernate.properties;

∙ persistence.xml.

Самый частый способ конфигурации: через аннотации и файл persistence.xml, что касается файлов hibernate.properties и hibernate.cfg.xml, то hibernate.cfg.xml главнее (если в приложение есть оба файла, то принимаются настройки из файла hibernate.cfg.xml). Конфигурация аннотациями, хоть и удобна, но не всегда возможна, например, если для разных БД или для разных ситуаций вы хотите иметь разные конфигурацию сущностей, то следует использовать xml файлы конфигураций.

По мимо этого хибернейт можно сконфигурировать с использованием SessionFactory или EntityManagerFactory.

При использовании JPA или Hibernate у вас есть два варианта:

∙ Вы можете загрузиться с помощью встроенного механизма Hibernate и создать SessionFactory.

∙ Или вы можете создать JPA EntityManagerFactory

Начальная загрузка через JPA должна быть предпочтительной. Кроме того, если вы использовали JPA, и вы вводили EntityManagerFactory через @PersistenceUnit аннотации:

|  |  |
| --- | --- |
| 1.  2. | @PersistenceUnit  private EntityManagerFactory entityManagerFactory; |

Вы можете легко получить доступ к базовому SessionFactory используя unwrap метод:

|  |  |
| --- | --- |
| 1.  2.  3. | SessionFactory sessionFactory =  entityManagerFactory.unwrap(  SessionFactory.class); |

То же самое можно сделать с JPA EntityManager. Если вы вводите EntityManager через @PersistenceContext аннотацию:

|  |  |
| --- | --- |
| 1.  2. | @PersistenceContext  private EntityManager entityManager; |

Вы можете легко получить доступ к базовому, Session используя unwrap метод:

|  |  |
| --- | --- |
| 1.  2. | Session session =  entityManager.unwrap(Session.class); |

Вывод

Т. о., вам следует загружать через JPA, использовать EntityManagerFactory и EntityManager и развертывать их только в связанных с ними интерфейсах Hibernate, когда вы хотите получить доступ к некоторым специфичным для Hibernate методам, которые недоступны в JPA, например, к извлечению объекта через его естественный идентификатор.

### На какой стороне инкрементится id — на стороне базы или хибернейта

GenerationType.IDENTITY — id инкрементится на стороне базы;

GenerationType.SEQUENCE — id инкрементится на стороне хибера, использует дополнительные селекты, чтобы запросить id.

### Что означает двусторонняя связь. Как это отобразится в коде

У нас есть таблица для каждой корзины и еще одна таблица для каждого товара. В одной корзине может быть много товаров, поэтому здесь у нас есть отображение «один ко многим».

То, как это работает на уровне БД, заключается в том, что у нас есть cart\_id в качестве первичного ключа в таблице корзины, а также cart\_id в качестве внешнего ключа в элементах .

И способ, которым мы делаем это в  коде, с @OneToMany .

Давайте сопоставим класс Cart объекту Items таким образом, чтобы он отражал отношения в БД:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | @Entity  @Table(name="CART")  public class Cart {  //...  @OneToMany(mappedBy="cart")  private Set<Items> items;  // getters and setters  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | @Entity  @Table(name="ITEMS")  public class Items {  //...  @ManyToOne  @JoinColumn(name="cart\_id", nullable=false)  private Cart cart;  public Items() {}  // getters and setters  } |

Мы также можем добавить ссылку на корзину в пунктах, используя @ManyToOne, что делает это двунаправленным отношением. Двунаправленный означает, что мы можем получить доступ к предметам из корзины, а также к корзине из предметов.

Свойство mappedBy — это то, что мы используем, чтобы сообщить Hibernate, какую переменную мы используем для представления родительского класса в нашем дочернем классе.

@OneToMany используется для определения свойства в классе Items, которое будет использоваться для сопоставления переменной mappedBy. Вот почему у нас есть свойство с именем «cart» в классе Items:

@ManyToOne связана с переменной класса Cart. Аннотация @JoinColumn ссылается на сопоставленный столбец.

### Что произойдет если не поставить mappedBy на @OneToOne @OneToMany — почему выгодно ставить mapped by на этих типах связей

mappedBy делает связь двунаправленной. Без mappedBy связь будет однонаправленной.

После того, как мы определили сторону-владельца отношений, Hibernate уже располагает всей информацией, необходимой для отображения этих отношений в нашей БД. Чтобы сделать эту ассоциацию двунаправленной, все, что нам нужно сделать, это определить сторону ссылки. Обратная или ссылочная сторона просто отображается на сторону-владельца.

Мы можем легко использовать mappedBy атрибут @OneToMany аннотацию, чтобы сделать это. Итак, давайте определим нашу сущность Employee:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | @Entity  @Table(name="ITEMS")  public class Items {  //...  @ManyToOne  @JoinColumn(name="cart\_id", nullable=false)  private Cart cart;  public Items() {}  // getters and setters  } |

Здесь значение mappedBy является именем атрибута сопоставления ассоциации на стороне владельца. Благодаря этому мы установили двустороннюю связь между нашими сотрудниками и сотрудниками электронной почты.

### Университеты и Студенты — @MаnyToMany — можно ли обойтись просто этой аннотацией без @JoinTable и @JoinColumn

Вроде как нет, т. к. нам нужна третья таблица для связи ManyToMany (связать первичный и внешний ключ).

### При аннотации @ManyToMany — ОБА ВЛАДЕЛЬЦЫ СВЯЗИ

Настройка каскадов этой связи немного сложнее т. к. связь по умолчанию является двунаправленной, и, более того, в этой связи зачастую невозможно выделить собственность и собственника, т. к. границы между ними сильно размыт. Получается картина, когда обе стороны могут передавать каскадные изменения друг на друга.

Важно не использовать на связи @ManyToMany CascadeType.All, т. к. последний включает в себя каскад на Remove, а этот каскад заставляет удалять хибернейт данные не только из смежной таблицы, но и дальше, из таблицы сущности, которая привязана к данной. А за этим, если к этой сущности были привязаны ещё несколько других — произойдет и их удаление из базы

### @ElementCollection

Определяет коллекцию экземпляров базового типа или встраиваемого класса. Должен быть указан, если коллекция должна отображаться с помощью таблицы коллекции.

### OrphanRemoval

@OrphanRemoval - управляет поведением осиротевшими сущностями.

OrphanRemoval: если мы вызовем setOrders(null), Order энтити будет удалена из БД автоматически.

Если каскад ремув, то setOrders(null), Order НЕ БУДЕТ УДАЛЕНА из БД автоматически.

Удаление сирот в отношениях

Когда целевой объект в отношении «один-к-одному» или «один-ко-многим» удаляется из отношения, часто желательно каскадно удалить операцию для целевого объекта. Такие целевые объекты считаются «сиротами», а атрибут orphanRemoval может использоваться для указания того, что потерянные объекты должны быть удалены. Например, если в заказе много позиций, и одна из них удалена из заказа, удаленная позиция считается сиротой. Если для orphanRemoval установлено значение true, объект позиции будет удален при удалении позиции из заказа.

### Почему есть Cascade Removal и orphanRemoval — есть еще один случай — когда есть разница? А при удалении разницы нет — в каком случае есть?

CascadeType определяет каскадные операции, которые применяются в элементе каскада аннотаций отношений.

Пример: позиция является частью заказа; если заказ удален, позиция также должна быть удалена. Это называется каскадным отношением удаления.

orphanRemoval — когда целевой объект в отношении один-к-одному или один-ко-многим удаляется из отношения.

Пример: если в заказе много позиций, и одна из них удалена из заказа, удаленная позиция считается сиротой. Если для orphanRemoval установлено значение true, объект позиции будет удален при удалении позиции из заказа.

Когда использовать:

Каскадное удаление удаляет все дочерние элементы при удалении родителя.

Таким образом, если вы удалите пользовательскую сущность, JPA удалит также все его фотографии.

### Метод unWrap()

JPA обеспечивает легкий доступ к API базовых реализаций. EntityManager и EntityManagerFactory обеспечивают разворачивают метод, который возвращает соответствующие классы реализации JPA. В случае Hibernate это Session и SessionFactory.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | Session session = em.unwrap(Session.class);  SessionFactory sessionFactory =  em.getEntityManagerFactory().unwrap(  SessionFactory.class); |

В первой строке я получаю текущий сеанс Hibernate от EntityManager. Поэтому я называю UnWrap метод на EntityManager и обеспечить сеанс класса в качестве параметра.

Вторая строка выглядит очень похоже. Я получаю EntityManagerFactory для текущего EntityManager и вызываю метод unwrap специфичный для Hibernate класс SessionFactory.

Эти классы предоставляют вам полный доступ к проприетарным функциям Hibernate, таким как поддержка Streams и Optional.

### Каким методом очищается кэш 1-го уровня

Это делается с помощью двух методов:

evict()

clear()

Здесь evict() используется для удаления конкретного объекта из кэша, связанного с сеансом, а метод clear() используется для удаления всех кэшированных объектов, связанных с сеансом.

# EntityManager

Интерфейс EntityManager — главный API для работы с JPA.

Основные операции:

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.

# Основные аннотации

## @Entity

Указывает класс как entity bean (объект с определенными условиями).

### Требования к Entity в JPA.

Entity класс обязан  
∙ быть помечен аннотацией @Entity или описан в XML файле,  
∙ содержать конструктор без аргументов (public или protected),  
∙ 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).

# @Embeddable и @Embedded (встраиваемый класс)

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);

## @Table

Используется для определения имени таблицы из БД, которая будет отображаться на entity bean.

## @Access

Определяет тип доступа. Обращение к атрибутам Entity как к полям класса (AccessType.FIELD) или как к свойствам класса (AccessType.PROPERTY).

## @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).

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).

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.

## @Basic

Указывает на простейший тип маппинга данных на колонку таблицы БД. Также в параметрах аннотации можно указать fetch стратегию доступа к полю и является ли это поле null или нет.

Самый простой тип сопоставления со столбцом БД. Базовая аннотация может быть применена к постоянному свойству или переменной экземпляра любого из следующих типов: примитивные типы Java, оболочки примитивных типов, String, BigInteger, BigDecimal, Date, Calendar, Time, Timestamp, byte[], Byte[], char[], Character[], enums, и любой другой тип, который реализует java.io.Serializable.

Использование базовой аннотации является необязательным для постоянных полей и свойств этих типов. Если базовая аннотация не указана для такого поля или свойства, то будут применяться значения по умолчанию базовой аннотации.

## @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 |

## @GeneratedValue

Задает стратегию создания основных ключей. Используется в сочетании с javax.persistence.GenerationType enum.

@OneToOne

Задает связь один-к-одному между двумя сущностными бинами. Соответственно есть другие аннотации @OneToMany, @ManyToOne и @ManyToMany.

## @Cascade

Определяет каскадную связь между двумя entity бинами. Используется в связке с org.hibernate.annotations.CascadeType.

# @PrimaryKeyJoinColumn

Определяет внешний ключ для свойства. Используется вместе с org.hibernate.annotations.GenericGenerator и org.hibernate.annotations.Parameter.

# @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.

# Три стратегии построения иерархии

∙ 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 потребуются большое количество отдельных sql запросов или использование UNION запроса для выборки всех классов иерархии.

# Маппинг enum

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

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

# Маппинг дат

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

# !!!Как смапить коллекцию примитивов

# Типы связей

OneToOne, OneToMany, ManyToOne и ManyToMany.

Каждая разделяется еще на два вида:

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

Unidirectional (ссылка устанавливается только с одной стороны).

# Владелец связи (mappedBy).

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

|  |  |
| --- | --- |
| 1. | @OneToMany(mappedBy = "nameColumn") |

У 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.

# Каскадирование

Действие над целевой сущностью 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.

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:  
∙ one(many)to many — lazy;  
∙ one(many)to one — eagle

26. Стратегии загрузки по умолчанию для всех видов связей, а также для аннотации @Basic и @Collection

EAGER для @Basic и ToOne

LAZY для @Collection и ToMany

Жизненный цикл сущности

new — бъект 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, и объект будет сохранен в базу при commit транзакции или в результате flush операций;

∙ managed → операция игнорируется, однако зависимые Entity могут поменять статус на managed, если у них есть аннотации каскадных изменений;

∙ removed → managed;

∙ detached → exception сразу или на этапе commit транзакции.

## remove()

∙ new → операция игнорируется, однако зависимые Entity могут поменять статус на removed, если у них есть аннотации каскадных изменений и они имели статус managed;

∙ managed → removed и запись объект в БД будет удалена при commit транзакции (также произойдут операции remove для всех каскадно зависимых объектов);

∙ removed → операция игнорируется;

∙ detached → exception сразу или на этапе commit транзакции.

## merge()

∙ detached → либо данные будут скопированы в существующий managed entity с тем же первичным ключом, либо создан новый managed в который скопируются данные

∙ new → будет создана новый managed entity, в который будут скопированы данные прошлого объекта;

∙ managed → операция игнорируется, однако операция merge сработает на каскадно зависимые Entity, если их статус не managed;

∙ removed → exception сразу или на этапе commit транзакции.

## refresh()

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

∙ new, removed, detached → exception.

## detach()

∙ managed, removed → detached;

∙ new, detached → операция игнорируется.

# 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).

# @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.

# @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

# Блокировки (оптимистические и пессимистические).

Блокировки — механизм, позволяющий параллельную работу с одними и теми же данными в БД.

Оптимистичный подход предполагает, что параллельно выполняющиеся транзакции редко обращаются к одним и тем же данным и позволяет им спокойно и свободно выполнять любые чтения и обновления данных. Но, при окончании транзакции, т. е. записи данных в базу, производится проверка, изменились ли данные в ходе выполнения данной транзакции и если да, транзакция обрывается и выбрасывается исключение.

Пессимистичный подход напротив, ориентирован на транзакции, которые постоянно или достаточно часто конкурируют за одни и те же данные и поэтому блокирует доступ к данным превентивно, в тот момент когда читает их. Другие транзакции останавливаются, когда пытаются обратиться к заблокированным данным и ждут снятия блокировки (или кидают исключение).

Оптимистичная блокировка делится на два типа: LockModeType.OPTIMISTIC (блокировка на чтение) и LockModeType.OPTIMISTIC\_FORCE\_INCREMENT (блокировка на запись).

Пессимистичная блокировка на 2 типа: LockModeType.PESSIMISTIC\_READ (данные блокируются в момент чтения) и LockModeType.PESSIMISTIC\_WRITE (данные блокируются в момент записи).

# 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.

# Criteria API

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().

# n + 1 select (описание и решения)

Допустим, у вас есть коллекция Car объектов (строк БД), и у каждого Car есть коллекция Wheel объектов (также строк). Другими словами, Car‑Wheel это отношение один-ко-многим.

Теперь предположим, что вам 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.