1 Введение

1.1 О проекте hibernate

Hibernate это проект, ставящий своей целью разработку open-source решения для организации слоя манипуляции данными объектно-ориентированного приложения, хранящимися в реляционной базе данных.
Данный проект стартовал в 2001 году, выпускается под лицензией LGPL. Его главным идеологом и архитектором является Gavin King, поддерживают проект корпорации jboss, ibm, oracle.

1.2 О персистентности

Идеологически данные (объекты) в приложении делятся на два типа:

- персистентные (persistent) – состояние данных объектов хранится в долговременном хранилище, они могут быть в любой момент восстановлены из этого хранилища или сохранены в него.
- временные (transient) – эти объекты создаются в ходе работы приложения и ограничены во времени жизни длительностью работы приложения.

Создатели hibernate предлагают выделять в приложении слой управления персистентными объектами. Основными функциями этого слоя являются:

- хранение, организация и получение структурированных данных
- поддержание целостности данных
- конкурентный доступ к данным и совместное использование данных

При создание этого слоя основными требованиями к нему являются прозрачность операций над персистентными объектами и понятный API. Вообще, самом простом случае, достаточно иметь реализацию четырех основных действий над персистентным объектом “create, read, update, delete” (CRUD). Но, конечно, hibernate содержит гораздо больше функциональности.

1.3 Об ORM

Существует несколько способов долговременного хранения объектных данных. Эти данные можно хранить в объектных базах, в сериализованном виде. Основным (номинальным) способом хранения объектных данных в hibernate является ORM (object-relationship mapping). ORM это способ хранения объектных данных в реляционной базе данных, при котором описывается соответствие (mapping) метаданных об объектной структуре приложения и реляционной структуры базы данных. К ORM-решению выдвигаются следующие требования:

- ORM должен предоставлять API для проведения CRUD операций;
- Язык или API составления запросов к хранилищу для получения объектов и свойств объектов;
- Средства описания маппинга метаданных;
- Возможность реализации механизмов оптимизации операций с персистентными объектами (dirty checking, «ленивая» загрузка и т.д.)

Все эти требования в hibernate реализованы.

2 Простой пример использования

Пример взят из документации Hibernate (http://www.hibernate.org/hib_docs/v3/reference/en/html/) .
Сначала описывается класс персистентного объекта

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    private void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

На основе его метаданных создается схема маппинга. Это – xml-файл. По соглашению, он дожжен лежать рядом с описываемым классом, иметь такое же название и расширение hbm.xml. В нашем случае, Cat.hbm.xml
Как берется тип по умолчанию

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

Примеры работы с персистентным объектом. Создание и сохранение объекта.

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

Запрос к базе данных с использованием языка Hibernate Query Language (HQL)

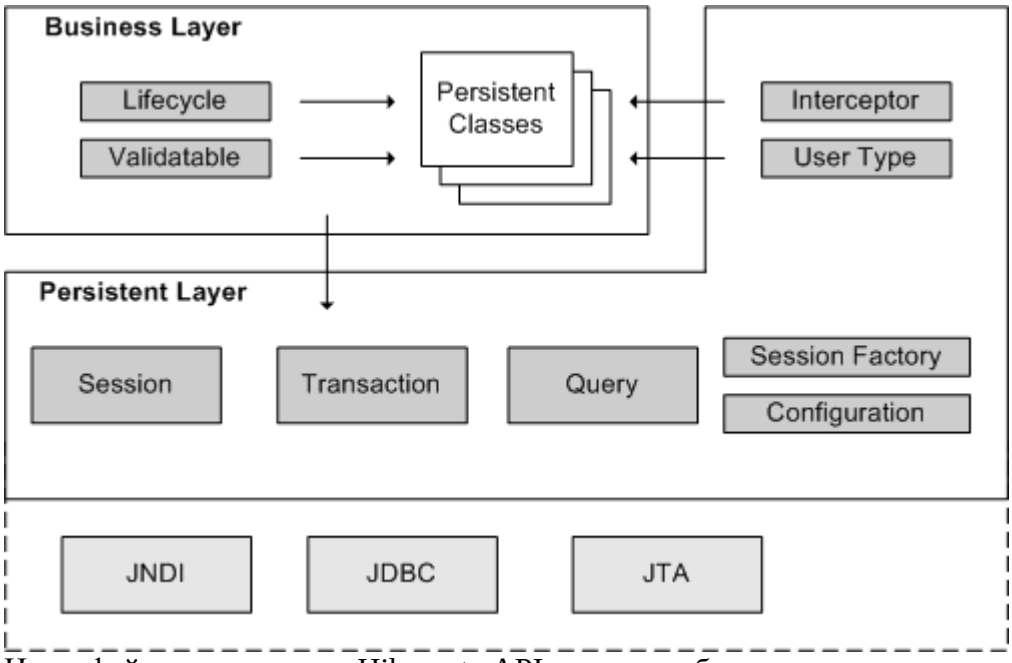
```
Transaction tx = session.beginTransaction();

Query query = session.createQuery("select Cat.name from Cat where sex =
:sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Создатели документации предлагают создавать отдельный класс как доступа с API hibernate. Привожу реализацию:
currentSession автоматически в HibernateFactory

```
import net.hibernate.*;
import net.hibernate.cfg.*;
```

```
public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

3 Общий взгляд на архитектуру и API Hibernate



Интерфейсы, входящие в Hibernate API можно разбить на следующие группы:

1. Интерфейсы, помогающие исполнять операции, входящие в CRUD, и вообще служащие для написания бизнес-логики. Это Session, Transaction, Query.
2. Интерфейсы, обеспечивающие конфигурацию hibernate. Это, прежде всего, Configuration.
3. Интерфейсы, обеспечивающие гибкую настройку маппингов. Это UserType, CompositUserType, IdentifierGenerator.
4. Интерфейсы, обеспечивающие обратную связь (callbacks) уровня персистенции и бизнес-логики. Это Lifecycle, Interceptor, Validatable.

3.1.1 Session

По аналогии с базами данных это нечто среднее между connection и transaction. Содержит функциональность persistent manager – менеджера персистентных объектов. Является кэшем объектов первого уровня (first level cache).

3.1.2 SessionFactory

Генератор сессий. Предполагается, что в приложении будет только одна фабрика сессий для всех нитей. Содержит метаинформацию по маппингам, кэш второго уровня (second level cache), кэш запросов и другую статическую информацию.

3.1.3 Configuration

Объект конфигурации конфигурирует и стартует Hibernate, создает SessionFactory.
Transaction
Интерфейс Transaction используется как абстракция для транзакций в ее различных реализациях – JDBC транзакция, JTA UserTransaction или Common Object Request Broker Architecture (CORBA) транзакция. По сути, транзакция имеет смысл группировки действий в источнике данных со своими стандартными свойствами – атомарности, согласованности, изолированности и долговременности (классические свойства atomicity, consistency, isolation, durability - ACID).

3.1.4 Query и Criteria

Как было заявлено выше, hibernate предоставляет язык и API создания запросов к хранилищу данных в метаданных персистентных классов. Query это интерфейс создания и исполнения запросов на языке HQL (hibernate query language), а Criteria позволяет формировать запросы, используя API.

3.1.5 Callback

Это интерфейсы, которые используются для создания реакций на действия с персистентными объектами и объектами hibernate API.

3.1.6 User Types

Интерфейсы для описания пользовательских типов метаданных, которые можно использовать в описаниях маппингов.

3.1.7 Интерфейсы расширения

- Primary key generation (IdentifierGenerator interface)
- SQL dialect support (Dialect abstract class)
- Caching strategies (Cache and CacheProvider interfaces)
- JDBC connection management (ConnectionProvider interface)
- Transaction management (TransactionFactory, Transaction, and TransactionManagerLookup interfaces)
- ORM strategies (ClassPersister interface hierarchy)

- Property access strategies (PropertyAccessor interface)
- Proxy creation (ProxyFactory interface)

4 Конфигурирование hibernate

Пример конфигурационного файла

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

   <session-factory>

        <!-- properties -->
        <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</pro
perty>
        <property
name="hibernate.connection.url">jdbc:oracle:thin:@//195.64.216.211:1521/devel
db.office0.naumen.ru</property>
        <property name="hibernate.connection.username">achernin</property>
        <property name="hibernate.connection.password">123</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>

        <property name="hibernate.connection.autoReconnect">true</property>
        <property name="hibernate.connection.pool_size">10</property>
        <property name="hibernate.connection.useUnicode">true</property>
        <property name="hibernate.connection.characterEncoding">UTF-
8</property>
        <property name="hibernate.show_sql">false</property>
        <property name="hibernate.jdbc.batch_size">100</property>
               <property
name="hibernate.cache.provider_class">org.hibernate.cache.OSCacheProvider</pr
operty>

        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- mapping files -->
        <mapping resource="
net/sf/hibernate/examples/quickstart/Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Элементы конфигурации:

- Диалект и параметры соединения (jdbc-соединения)
- Параметры пула соединений (размер, таймауты), библиотека пула (встроенная, c3po, proxool, apache dbcp)
- Параметры источника соединений
- Различные параметры
    - Общие параметры
    - Параметры каша второго уровня
    - Параметры транзакций
- Список ресурсов

Дополнительно настраивается ведение логов (журнализация). В hibernate используется библиотека log4j, ее настройки находятся в отдельном файле.
Кроме того, есть набор настроек, которые используются для интеграции hibernate в состав сервера приложений J2EE.

5 Персистентные классы и основы маппинга

Персистентным, т.е. сохраняемым в базу данных, может быть объявлен любой класс. Его не нужно наследовать от каких-либо суперклассов или интерфейсов, но он должен подчиняться простым правилам, известным как Plain Old Java Object (POJO).
Классы POJO должны содержать конструктор без параметров, bean-like свойства, часть которых может содержать ссылки на другие POJO и служебные методы. Также рекомендуется делать POJO классы сериализуемыми (включать интерфейс Serializable), переопределять в них вычисление хэш-функции и сравнение (hashCode и equals).
Для класса описывается маппинг– создается xml-описание. Пример из документации:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats">

                <id name="id">
                        <generator class="native"/>
                </id>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
```

```
        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

- Основные тэги:
- hibernate-mapping – тэг документа. Имеет набор свойств, в основном – дефолтные значение;
- class – тэг класса. Обязательным является только имя класса. Из остальных атрибутов выделим table, proxy, select-before-update, polymorphism, lazy, abstract.
- Id – тэг идентификатора. Идентификатор это, по сути, первичный ключ записи. Может быть как комбинацией полей объекта (естественный первичный ключ), так и генерируемым значением. Hibernate предоставляет набор алгоритмов генерации ключей – числовых и строковых, дает возможность использовать автоинкрементные поля базы данных или разработать собственный генератор.
- Property – тэг свойства. Основные атрибуты – name, type (обязательные), column, not-null, unique
- version – используется для сохранения версии объекта. Используется в режиме длинных транзакций для операции attach.
- timestamp – тэг времени последнего изменения объекта. Необходим в том случае, когда на объекты данного класса могут накладываться блокировки.

6 Персистентные коллекции

Hibernate позволяет работать не только с простыми свойствами персистентного объекта, но и со свойствами-коллекциями. Есть возможность выполнять все операции CRUD с коллекциями всех типов – набор (java.util.Set), список (java.util.List), словарь (java.util.Map).
При описании коллекции обязательно должен быть указано ключевое поле (`<key>`), по которому будут различаться элементы коллекции.
Элементами коллекции могут быть объекты любых типов, включая базовые типы, пользовательские типы, персистентные сущности и компоненты. Элементы в коллекции могут храниться как «по значению», так и «по ссылке». По значению хранятся объекты простых типов, по ссылке – коллекции персистентных сущностей. Элементы коллекции могут быть описаны тэгами `<element>`, `<composite-element>`, `<one-to-many>`, `<many-to-many>` or `<many-to-any>`.
Для коллекций типа список и словарь указывается поле индекса. Для списков индекс используется для сортировки элементов (должен содержать целочисленное значение), для словарей – ключ словаря (им может быть любой объект). Индексы могут быть описаны тэгами `<index>`, `<index-many-to-many>`, `<composite-index>` or `<index-many-to-any>`.

6.1 Пример маппинга коллекции

Коллекции могут быть объявлены тэгами `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` и `<primitive-array>`. Приведем пример синтаксиса тэга map (словарь), для других видов коллекций может быть приведено примерно такое же описание.

```
<map
    name="propertyName"                                    (1)
    table="table_name"                                     (2)
    schema="schema_name"                                   (3)
```

```
    lazy="true|false"                                                (4)
    inverse="true|false"                                             (5)
    cascade="all|none|save-update|delete|all-delete-orphan"          (6)
    sort="unsorted|natural|comparatorClass"                          (7)
    order-by="column_name asc|desc"                                  (8)
    where="arbitrary sql where condition"                            (9)
    outer-join="true|false|auto"                                     (10)
    batch-size="N"                                                   (11)
    access="field|property|ClassName"                                (12)
>

    <key column="column_name"/>                                      (13)
    <index
        column="column_name"                                         (14)
        type="typename"                                              (15)
    />
    <element
        column="column_name"                                         (16)
        type="typename"                                              (17)
    />

</map>
```

(1) name имя свойства.
(2) table (опциональное – по умолчанию имя свойства) имя таблицы для коллекции (не используется в случае связи один ко многим one-to-many).
(3) schema (опционально) имя схемы, в которой создается таблица. Перекрывает имя схемы, указанное в корневом элементе.
(4) lazy (опционально – по умолчанию false) определяет использование ленивой инициализации (не используется для массивов array)
(5) inverse (опционально – по умолчанию false) помечает коллекцию как обратную (inverse) часть двунаправленной связи
(6) cascade (опционально – по умолчанию пустое none) операции, которые будут выполняться каскадно с подчиненными сущностями.
(7) sort (опционально – по умолчанию natural) возможность указать компаратор для сортировки значений (если это позволяет тип коллекции)
(9) where (опционально) позволяет указать условие (SQL where), ограничивающее набор данных
(10) outer-join (опционально – по умолчанию true) управляет функцией выборки коллекции в одном запросе с объектом-владельцем
(11) batch-size (опционально – по умолчанию 1) определяет «batch size» (размер пакета?) при ленивой загрузке элементов этой коллекции.
(12) access (опционально – по умолчанию берется настройка для свойства) стратегия доступа к значению.
(13) column (обязательное) имя колонки внешнего ключа
(14) column (обязательное) имя колонки с индексом коллекции
(15) type (опционально – по умолчанию integer) тип индекса коллекции
(16) column (обязательное) имя колонки с элементами коллекции
(17) type (обязательное) тип элементов коллекции

6.2 Ленивая инициализация (lazy initialization)

Все коллекции (кроме массивов) могут «инициализироваться лениво», т.е. объекты в них загружаются из базы данных в момент первого обращения к коллекции (а не в момент загрузки из базы данных самого объекта). Эта загрузка происходит прозрачно для пользователя.

Плюсы: Ускорение загрузки объекта – владельца коллекции, лишние данные не загружаются, если они не нужны.
Минусы: Могут возникать ошибки при работе с detached объектами, т.е. если происходи обращение к неинициализированной коллекции объекта после закрытия сессии, в которой объект был создан. Соответственно, при обращении к коллекции объекта должна предусматриваться возможность такой коллизии и объект должен привязываться к новой (другой) сессии.

6.3 Примеры коллекций

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

В этом примере у класса eg.Parent есть свойство children типа набор (Set).
Маппинг может быть описан в виде связи один ко многим

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

В результате получается следующая конфигурация базы данных

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255),
parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

Если родитель *обязателен*, рекомендуется двунаправленная связь один ко многим

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.Parent" column="parent_id" not-
null="true"/>
    </class>

</hibernate-mapping>
```

Обратите внимание на инструкцию NOT NULL в связи один ко многим, которая преобразуются в аналогичную инструкцию для колонки внешнего ключа в подчиненной таблице

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Для случая, когда у eg.Child может быть несколько родителей (eg.Parent), рекомендуется коллекция с использованием связи типа многие ко многим.

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
```

```
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

7 Ссылочная связность

Ссылки между POJO оформляются как пара взаимодополняющих методов в обоих связанных классах.
Например, если класс Категория (Category) ссылается на вложенные категории, то создается поле parentCategory типа Category и поле childCategories типа Set. Работа с ними выглядит так:

```
import java.util.Set;
import java.util.HashSet;
import java.io.Serializable;

public class Category implements Serializable
{

    public Category() {
    }

    private Set childCategories = new HashSet();
    private Category parentCategory = null;

    public Set getChildCategories() {
        return childCategories;
    }

    public void setChildCategories(Set childCategories) {
        this.childCategories = childCategories;
    }

    public Category getParentCategory() {
        return parentCategory;
    }

    public void setParentCategory(Category parentCategory) {
        this.parentCategory = parentCategory;
    }

    public static void addChildCategory(Category parentCategory, Category
childCategory)
    {
        parentCategory.getChildCategories().add(childCategory);
        childCategory.setParentCategory(parentCategory);
    }

}
```

Как известно, связь может быть трех типов – один к одному, один ко многим и многие ко многим. В данном случае, связь один ко многим – у одной родительской категории несколько дочерних категорий. И обратно – если у одной категории есть родительская категории, то только одна.
Также известны правила связывания сущностей между собой в реляционной базе данных. В случае «один к одному» и «один ко многим» создается констрэйнт (реляционная связь) между таблицами, в случае «многие ко многим» создается связующая таблица. Очевидно,

в любом случае для создания связей требуется, чтобы у связываемых сущностей были заданы первичные ключи.
Типы связей в том виде, в котором они даны в документации Hibernate

7.1 Однонаправленные связи с использованием внешнего ключа

7.1.1 Многие к одному

Это наиболее часто встречающийся вариант

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null )
create table Address ( addressId bigint not null primary key )
```

7.1.2 Один к одному

В этом случае связь «один к одному» моделируется как частный случай связи «многие к одному». Накладываются дополнительные ограничения на уникальность поля – внешнего ключа.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null unique )
create table Address ( addressId bigint not null primary key )
```

В этом примере связь «один к одному» делается на уровне совпадающего первичного ключа (в базе данных будет создана ссылка также для первичного ключа)

```
<class name="Person">
    <id name="id" column="personId">
```

```
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2 Однонаправленные связи с использованием связующей таблицы

7.2.1 Один ко многим

Этот вариант является предпочтительной реализацией связи этого типа. Связь «многие ко многим» ограничивается до «одного ко многим» условием уникальности адреса.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null
primary key )
create table Address ( addressId bigint not null primary key )
```

7.2.2 Многие к одному

Такая реализация рекомендуется в случае необязательности связи.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
```

```
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId
bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.3 Многие ко многим

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

7.3 Двунаправленные связи с использованием внешнего ключа

7.3.1 Один ко многим / многие к одному

Данный вариант связи считается наиболее распространенным

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
```

```
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null )
create table Address ( addressId bigint not null primary key )
```

Для того, чтобы получать сортированный список объектов, нужно сделать обязательным внешний ключ (свойство key) и оставить возможность редактировать связи со стороны коллекции (выставив запрет изменения и добавления набора связей `update="false"`, `insert="false"`).

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

7.3.2 Один к одному

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint
not null unique )
create table Address ( addressId bigint not null primary key )
```

По аналогии с предыдущим примером для этого типа связи – связь «один к одному» на основе ссылки в первичном ключе.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.4 Двунаправленные связи с использованием связующей таблицы

7.4.1 Один ко многим / многие к одному

Обратите внимание: inverse="true" может указываться в подчиненной сущности.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null primary key )
create table Address ( addressId bigint not null primary key )
```

7.4.2 Многие ко многим

```
<class name="Person">
```

```
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not
null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

8 Маппинг компонентов

Под компонентом понимается поле, которое имеет типом какой-то класс, который не хочется делать персистентным. Тогда данные этого класса включаются (агрегируются) в данные класса владельца.

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
```

```
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

В данном примере класс Person агрегирует поле класса Name.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

Свойства класса-компонента (Name) будут храниться в одной таблице со свойства (Person), а при инициализации будут создаваться оба объекта с агрегацией.

9 Маппинг наследования

9.1 Три стратегии

Стратегия маппинга наследования заключается в том, как данные класса распределяются по таблицам в базе данных. Кроме того, если нескольких классов имеют общий суперкласс, то информация о маппинге данных базового класс для всех подклассов является общей. Хочется один раз написать метаинформацию для базового класса и больше ее не дублировать.

Hibernate поддерживает три стратегии маппинга наследования классов

- таблица для иерархии (table per class hierarchy)
- таблица для подкласса (table per subclass)
- таблица для класса (table per concrete class)

Рассмотрим на примере класс Платеж (Payment) , который имеет трех наследников Платеж_Крединой_Картой (CreditCardPayment), Платеж_наличными (CashPayment), Платеж_Чеком (ChequePayment).

9.1.1 Таблица для иерархии (table per class hierarchy)

```
<class name="Payment" table="PAYMENT">
```

```
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

В базе данных будет создана одна таблица (PAYMENT), в ней будут созданы поля для всех четырех классов – общие (базового класса) и для каждого из классов-наследников. Дополнительно будет создано поле дискриминатора, которое будет заполняться кодом в зависимости от класса сохраняемого объекта. По этому же полю будет определятся класс при загрузке объекта любого из классов. В конкретном картеже будут заполнены поля данного класса, поля других классов будут нулевыми (null).

9.1.2 Таблица для подкласса (table per subclass)

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

В этом случае создается таблица для каждого из классов с полями только данного класса (т.е. 4 таблицы). Таблицы наследников ссылаются на таблицу базового класса (ссылка по первичному ключу).

9.1.3 Таблица для класса (table per concrete class)

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
```

```
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE PAYMENT">
        ...
    </union-subclass>
</class>
```

В этом случае создается по таблицу на каждый подкласс, каждая из них включает поля данного подкласса и базового класса. Данная стратегия хороша в том случае, если базовый класс – абстрактный. Иначе будет создана 4-я таблица для хранения объектов базового класса (т.е. не расширенного).

9.1.4 Таблица для подкласса с дискриминатором

Hibernate позволяет комбинировать стратегии. Например, можно распределить данные по 4-м таблицам (т.е. прийти к стратегии «таблица для подкласса»), но использовать дискриминатор типа в таблице базового класса.

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

Возможны и другие варианты микширования.

10 Работа с объектами

10.1 Состояния персистентных объектов

Hibernate определяет следующие состояния объектов:

- *Временный (Transient)* – это объект, который был создан, но еще не ассоциирован с сессией Hibernate. Этот объект или не является персистентным, или создан просто

вызовом оператора new. Соответственно, у него нет представления в базе данных. Для hibernate этот объект не существует, его жизненный цикл контролируется приложением и удален он будет сборщиком мусора.

- *Персистентный (Persistent)* – персистентный объект имеет представление в базе данных и идентификатор. Он может быть сохраненным или загруженным и привязан к конкретной сессии (hibernate session). Hibernate отслеживает все изменения в этом объекте и синхронизирует его состояние с базой данных после завершения работы с объектом.
- *Отделенный (Detached)* – это объект, который изначально был персистентным, но работа с ним продолжается после закрытия сессии, к которой он был привязан.

10.2 Создание персистентных объектов

Вновь созданный объект является временным (*Transient*). После сохранения он прикрепляется к сессии и становится персистентным.

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

В процессе сохранения для объекта генерируется уникальный идентификатор.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

В этом примере показана работа со свойством-коллекцией. Заметьте, в этом случае уникальный идентификатор назначается напрямую, без использования генератора.

10.3 Загрузка объекта

Вызовом метода load() сессии можно загрузить (получить) персистентный объект по идентификатору.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

Как вариант, можно загрузить содержимое по ключу в существующий объект

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Тут важно заметить, что load() может возвращать не сам объект, а его неинициализированный прокси без загруженных данных и не обратиться к базе данных за ними до тех пор, пока не будет вызван любой из методов объекта. Это поведение очень важно в том случае, если объект нужен, например, для создания ссылки на него.

Для получения объекта по идентификатору лучше использовать метод get(), если вы не уверены в том, что объект с таким идентификатором существует. Данный метод сразу обращается в базу данных и возвращает null, если искомый объект не найден.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Еще один вариант получение объекта – получение с наложением блокировки

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Удаление объектов проводится вызовом метода delete() у сессии.

```
sess.delete(cat);
```

Объекты можно сохранять не по одному, а вызвать сохранение сразу всех изменений, сделанных в рамках указанной сессии.

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

При флэше (flush) сессии действия выполняются в следующем порядке:

1. все сохранения объектов
2. все изменения объектов
3. все удаления коллекций
4. все действия с элементами коллекций (удаления, добавления, изменения)
5. все добавления коллекций
6. все удаления объектов в порядке вызова Session.delete()

11 The Hibernate Query Language (HQL)

HQL это объектно-ориентированный язык запросов, позволяющий извлекать из базы данных сохраненные объекты и их свойства. HQL-запросы транслируются в sql с учетом текущего диалекта и исполняются, а результаты исполнения интерпретируются с учетом маппинга.
Приведем примеры запросов.
Выборка объектов по классу. В большинстве запросов необходимо указывать алиасы для объектов.

```
from Cat as cat
```

Выборка объектов с использованием инструкции join. Возможен join всех типов, определенных в ANSI SQL: inner join, left outer join, right outer join, full join. Могут быть указаны ограничения в присоединяемой таблице (сущности).

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten;
```

```
from Cat as cat left join cat.mate.kittens as kittens;

from Formula form full join form.parameter param;

from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

Выборка свойств объектов. Если в результатах одно свойство, то в результате объект известного класса. Если несколько свойств, то Object[], или может быть инстанцирован объект.

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr;

select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr;

select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr;
```

Можно использовать агрегирующие функции. Доступные ф-ции: avg(...), sum(...), min(...), max(...), count(*), count(...), count(distinct ...), count(all...) . Можно использовать эти функции совместно с группировкой.

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat;

select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight;
```

Запросы с полиформизмом. Можно указать класс или интерфейс требуемых объектов, и будут подобраны те персистентные сущности, которые удовлетворяют этому условию.

```
from java.lang.Object o;

from Named n, Named m where n.name = m.name;
```

Запросы с условием. Все как в sql.

```
from Cat as cat where cat.name='Fritz';

from Foo foo
where foo.bar.baz.customer.address.city is not null;
```

Использование выражений. Набор выражений включает заданные в ANSI SQL, но расширяется. Необходимо учитывать, как функции исполняются в sql или в движке hql. Полный набор функций – в документации hibernate.

```
from DomesticCat cat where cat.name between 'A' and 'B';
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' );

from Order order where order.items[0].id = 1234;
```

Запросы с сортировкой результатов.

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate;
```

Запросы с условиями на результат группировки (инструкцией having)

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK);
```

Подзапросы.

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
);

from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
);
```

Для исполнение hql-запросов создается объект типа Query, параметризуется и исполняется. Текст запросов может содержать параметры, которые заполняются перед исполнением запроса.

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and
foo.size=:size");
q.setProperty("name", "Fritz");
q.setProperty("size", new Integer(5));
List foos = q.list();
```

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

12 Запросы Criteria

Criteria это API, который hibernate предоставляет для программного конструирования запросов.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

Самый простой вариант использования – создается объект Criteria для класса, потом на объекты этого класса начинают накладываться условия.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
```

```
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

В классе Restrictions есть методы создания ограничений всех типов, поддерживаемых ANSI SQL. Ограничения могут объединяться в логические выражения.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Ограничения могут быть написаны на SQL (в формате Native SQL – см. ниже).

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)",
"Fritz%", Hibernate.STRING) )
    .list();
```

В запросах может происходить обращение к свойствам класса – как в условиях, так и в описании требуемого набора данных.

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" }
) )
    .list();
```

На результаты может накладываться сортировка

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

Могут использоваться связи между сущностями.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();

List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

Для работы с агрегирующими функциями и группировками используется объект Projection.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();

List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

13 Native SQL

Hibernate позволяет описывать запросы на SQL с использованием метаинформации о маппинге.

```
List cats = sess.createSQLQuery("select {cat.*} from cats cat")
    .addEntity("cat", Cat.class)
    .list();

List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where
kitten.mother = cat.id"
    )
    .addEntity("cat", Cat.class)
    .addJoin("kitten", "cat.kittens")
    .list();

String sql = "select cat.originalId as {cat.id}, " +
    "cat.mateid as {cat.mate}, cat.sex as {cat.sex}, " +
    "cat.weight*10 as {cat.weight}, cat.name as {cat.name} " +
    "from cat_log cat where {cat.mate} = :catId"

List loggedCats = sess.createSQLQuery(sql)
    .addEntity("cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

Со всеми возможностями замен можно ознакомиться по документации hibernate.

14 Управление транзакциями

При выборе стратегии управления транзакциями и сессиями нужно учитывать, что должна облегчить получение персистентных объектов, но уменьшить вероятность коллизий – т.е. несогласованного изменения одних и тех же персистентных объектов разными пользователями.
Для выполнения первого условия желательно удлинить время жизни персистентного объекта в системе. То есть передавать их между пользовательскими сессиями, желательно, привязанными к сессии.
Выполнение второго условия требует коротких атомарных транзакций и коротких сессий.

Если не вдаваться в подробности, разработчики и пользователи hibernate рекомендуют стратегию session-per-user-request.
В системе создается один объект hibernate session factory. Он является «тяжелым» (на его создание уходит много времени/ресурсов), но он хорошо работает в режиме многопоточности. Объект Session (сессия) является гораздо более легким объектом, но при длительном сроке жизни с ней начинаются проблемы – она не потокобезопасна (threadsafe), ее размер увеличивается по мере загрузки новых объектов из базы данных (за счет кэша первого уровня). В терминах web-приложения предлагается создавать сессии на каждое пользовательское обращение. В рамках сессии предлагается делать несколько транзакций, в которые объединять несколько логически связанных операций с персистентными объектами (что обеспечит атомарность транзакции). Использование режима auto-commit – флэш сессии после каждой операции с объектами – не рекомендуется.

15 Прерывания (Interceptors)

Прерыватели реализуют интерфейс `Interceptor , который содержит методы для обратных вызовов (callbacks) при событиях загрузки, сохранения, изменения или удаления объектов.`

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            return true;
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
```

```
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            return true;
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " +
updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Прерыватель может быть помещен в сессию

```
Session session = sf.openSession( new AuditInterceptor() );
```

или в фабрику сессий

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

16 Увеличение производительности

16.1 Batch processing

Есть возможность выполнять hql-запросы в dml-стиле (data manipulation language, sql операции `INSERT`, `UPDATE`, `DELETE`).

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name =
:oldName";
// or String hqlUpdate = "update Customer set name = :newName where name =
:oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
```

```
session.close();
```

Практически, таким образом формируется один sql-запрос, модифицирующий группу объектов. Безусловно, это ускоряет выполнение требуемой модификации. С другой стороны, могут возникнуть проблемы из-за того, что изменения объектов проведены без выполнения программных обработчиков изменения объекта. Могут возникнуть проблемы с кешем второго уровня и блокировками.

16.2 Стратегии получения данных

Использование различных стратегий манипулирования данными является одним из основных способов ускорения приложения. Разработчики выделяют следующие стратегии:

- *Join fetching* - Hibernate получает связанные (с сущностью) объекты или коллекции в том же sql-запросе (селекте).
- *Select (lazy) fetching* – для получения связанного объекта или сущности используется еще один Select.
- *Subselect fetching* – вторым запросом получаются все связанные объекты или все коллекции для всех объектов из предыдущего запроса.
- *Batch fetching* – оптимизационная стратегия для запросов получения объектов. В запросе получаются не сами объекты и связанные с ним объекты и коллекции, а их идентификаторы.

16.3 Кэш второго уровня

При настройке hibernate может быть указана библиотека, используемая для кэширования объектов на уровне session factory, т.е. между сессиями. Для каждого персистентного класса при настройке маппинга может быть запрошено тестирование и указана стратегия тестирования.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

4 вида стратегий - read-only, nonstrict-read-write, read-write, transactional

- read only – только чтение. Используется, если объекты данного класса только читаются из базы данных, но не модифицируются.
- read-write – чтение и запись. При первой загрузке объект попадает в кэш, где храниться в сериализованном виде, при каждом изменении – перезаписывается.
- nonstrict-read-write – используется, если приложению требуется часто модифицировать данные.
- transactional - используется в системах с распределенным кэшированием

Библиотеки и поддерживаемые ими стратегии:

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		

JBoss TreeCache	yes			yes