Наследование в Hibernate: выбор стратегии

<https://habr.com/ru/post/337488/>

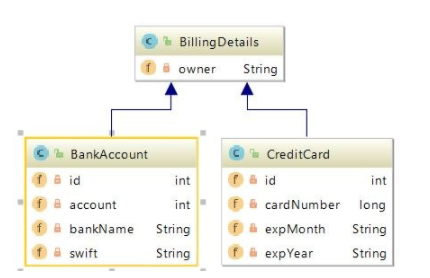
Наследование является одним из основных принципов ООП. В то же время, значительное количество корпоративных приложений имеют в своей основе реляционные базы данных  
  
Главное противоречие между объектно-ориентированной и реляционной моделями заключается в том, объектная модель поддерживает два вида отношений («is a» — “является”, и «has a» — “имеет”), а модели, основанные на SQL, поддерживают только отношения «has a».  
  
Иными словами, **SQL не понимает наследование типов и не поддерживает его**.  
  
Поэтому на этапе построения сущностей и схемы БД одной из главных задач разработчика будет выбор оптимальной стратегии представления иерархии наследования.  
  
Всего таких стратегий 4:  
  
**1)**Использовать одну таблицу для каждого класса и полиморфное поведение по умолчанию.  
  
**2)** Одна таблица для каждого конкретного класса, с полным исключением полиморфизма и отношений наследования из схемы SQL (для полиморфного поведения во время выполнения будут использоваться UNION-запросы)  
  
**3)** Единая таблица для всей иерархии классов. Возможна только за счет денормализации схемы SQL. Определять суперкласс и подклассы будет возможно посредством различия строк.  
  
**4)** Одна таблица для каждого подкласса, где отношение “is a” представлено в виде «has a», т.е. – связь по внешнему ключу с использованием JOIN.  
  
Можно выделить 3 главных фактора, на которые повлияет выбранная вами стратегия:  
  
**1)** Производительность (мы используем “hibernate\_show\_sql”, чтобы увидеть и оценить все выполняемые к БД запросы)  
  
**2)** Нормализация схемы и гарантия целостности данных (не каждая стратегия гарантирует выполнение ограничения NOT NULL)  
  
**3)**Возможность эволюции вашей схемы  
  
Под катом каждая из этих стратегий будет рассмотрена подробно, с указанием преимуществ и недостатков, а также будут даны рекомендации по выбору стратегии в конкретных случаях.

Пара слов от автора и инструкция по работе с примерами

Данная статья является выжимкой из книги «Java Persistance with Hibernate». Ее авторы — основатель проекта Hibernate Гэвин Кинг (Gavin King) и член команды разработчиков Hibernate Кристиан Баэур (Christian Bauer).  
Летом 2017 она была переведена и издана на русском языке.  
  
Я постарался упростить изложение материала, а также работу с примерами. Испытывая сильную нелюбовь к примерам, с которыми для запуска нужно возиться час, я стремился сделать работу с ними в этой статье максимально удобной:  
— Весь Java-код вы можете просто скопировать в свою IDE. Все изменения Java-кода при переходе от одной стратегии к другой указаны в спойлерах, поэтому при переходе к новой стратегии старый код класса можно просто удалить и скопировать новый. Классы Main и HibernateUtil останутся без изменений, и будут работать при рассмотрении всех примеров.  
  
— В спойлерах к каждой стратегии вы также найдете скрипты для создания всех таблиц БД. Поэтому после того, как вы разобрали очередную стратегию, можно просто дропнуть все таблицы — в следующем разделе вы найдете актуальные скрипты для создания новых.  
  
Код написан с использованием Java 1.7, Hibernate5 и PostgreSQL9  
  
Приятного прочтения!

Стратегия 1

Одна таблица для каждого класса

Ситуация:  
  
Мы решили затмить славу eBay и создаем для этой цели свое приложение интернет-аукциона. Каждый User может делать ставки, и в том случае если его ставка оказалась самой крупной – совершить оплату онлайн.  
  
Собственно, процесс оплаты мы и будем рассматривать в качестве модели данных.  
User может совершить оплату двумя способами: при помощи банковской карты, или посредством реквизитов банковского счета.  
  
Диаграмма классов представлена ниже:  
  


Java-код для запуска примера

pom.xml:

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.hiber.jd2050</groupId>

<artifactId>hiberLearn</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.7</java.version>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<target>${java.version}</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencies>

<!-- PostgreSQL -->

<dependency>

<groupId>postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.0-801.jdbc4</version>

</dependency>

<!-- Hibernate-JPA-2.1-API -->

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.1-api</artifactId>

<version>1.0.0.Final</version>

</dependency>

<dependency>

<groupId>javax.transaction</groupId>

<artifactId>jta</artifactId>

<version>1.1</version>

</dependency>

<!-- Hibernate-core -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.0.5.Final</version>

</dependency>

</dependencies>

</project>

hibernate.cfg.xml

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="connection.url">jdbc:postgresql://localhost:5432/dobrynin\_db</property> <!-- BD Mane -->

<property name="connection.driver\_class">org.postgresql.Driver</property>

<property name="connection.username">postgres</property>

<property name="connection.password">filyaSl9999</property>

<property name="dialect">org.hibernate.dialect.PostgreSQL9Dialect</property> <property name="hbm2ddl.auto">create-drop</property> <!-- create / create-drop / update -->

<property name="show\_sql">true</property> <!-- Show SQL in console -->

<property name="format\_sql">true</property> <!-- Show SQL formatted -->

<property name="hibernate.current\_session\_context\_class">thread</property>

<mapping class="CreditCard"/>

<mapping class="BankAccount"/>

<mapping class="BillingDetails"/>

</session-factory>

</hibernate-configuration>

import javax.persistence.\*;

@MappedSuperclass

public abstract class BillingDetails {

private String owner;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "CREDIT\_CARD")

public class CreditCard extends BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

@Column(name = "card\_number")

private int cardNumber;

@Column(name = "exp\_month")

private String expMonth;

@Column (name = "exp\_year")

private String expYear;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "BANK\_ACCOUNT")

public class BankAccount extends BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

private int account;

@Column(name = "bank\_name")

private String bankName;

private String swift;

Constructor, Getter, Setter, toString()

}

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class HibernateUtil {

public static SessionFactory getSessionFactory() {

return new Configuration().configure().buildSessionFactory();

}

}

Класс Main с методом main():

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.Transaction;

import java.util.\*;

public class Main {

public static void main(String[] args) throws Exception {

CreditCard creditCard = new CreditCard();

creditCard.setCardNumber(44411111);

creditCard.setExpMonth("Jan");

creditCard.setExpYear("2017");

creditCard.setOwner("Bill Gates");

BankAccount bankAccount = new BankAccount();

bankAccount.setAccount(111222333);

bankAccount.setBankName("Goldman Sachs");

bankAccount.setSwift("GOLDUS33");

bankAccount.setOwner("Donald Trump");

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session;

Transaction transaction = null;

try {

session = sessionFactory.getCurrentSession();

transaction = session.beginTransaction();

session.persist(creditCard);

session.persist(bankAccount);

transaction.commit();

} catch (Exception e) {

transaction.rollback();

throw e;

}

Session session1;

Transaction transaction1 = null;

try {

session1 = sessionFactory.getCurrentSession();

transaction1 = session1.beginTransaction();

List billingDetails = session1.createQuery("select bd from BillingDetails bd").list();

for (int i = 0; i < billingDetails.size(); i++) {

System.out.println(billingDetails.get(i));

}

} catch (Exception e) {

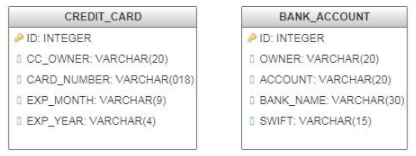
transaction1.rollback();

throw e;

}

}

}

Классы BankAccount и CreditCard наследуются от общего абстрактного предка BillingDetails. Как видно из схемы, несмотря на похожий функционал, их состояния существенно отличаются: для карты нам важны номер и срок действия, а для банковского счета – поля реквизитов.  
  
Родительский класс хранит только общую для всех потомков информацию о владельце.  
Кроме того, туда можно вынести, например, поле Id вместе с типом генерации (в данном случае мы обошлись без этого).  
  
Схема нашей БД для первой стратегии будет выглядеть так:  
  
  
Запросы для создания таблиц:

CREDIT\_CARD

create table credit\_card

(

id serial not null

constraint bank\_account\_pkey

primary key,

cc\_owner varchar(20) not null,

card\_number integer not null,

exp\_month varchar(9) not null,

exp\_year varchar(4) not null

);

BANK\_ACCOUNT

create table bank\_account

(

id serial not null

primary key,

owner varchar(20),

account integer not null,

bank\_name varchar(20) not null,

swift varchar(20) not null

);

Полиморфизм в данном случае будет неявным. Каждый класс-потомок мы можем отразить с помощью аннотации [Entity](https://habrahabr.ru/users/entity/).  
  
**ВАЖНО!**Свойства суперкласса по умолчанию будут проигнорированы. Чтобы сохранить их в таблицу конкретного подкласса, необходимо использовать аннотацию @MappedSuperClass.  
  
Отображение подклассов не содержит ничего необычного. Единственное, на что следует обратить внимание – возможно, незнакомая для некоторых аннотация @AttributeOverride.  
Она используется для переименования столбца в таблице подкласса, в том случае если названия у таблицы предка и таблицы потомка не совпадают (в нашем случае – чтобы «owner» из BillingDetails маппился (переименовывался) на CC\_OWNER в таблице CREDIT\_CARD).  
  
Главная проблема при использовании данной стратегии заключается в том, что использовать полиморфные ассоциации в полной мере будет невозможно: обычно они представлены в БД в виде доступа по внешнему ключу, а у нас попросту нет таблицы BILLING\_DETAILS. А поскольку каждый объект BillingDetails будет в приложении связан с конкретным объектом User, то каждой из таблиц-«потомков» нужен будет внешний ключ, ссылающийся на таблицу USERS.  
  
Кроме того, проблемой также будут и полиморфные запросы.  
  
Попробуем выполнить запрос

SELECT bd FROM BillingDetails bd

Для этого (здесь и далее) просто запустите метод main().  
  
В данном случае он будет выполнен следующим образом:  
Hibernate:

select

bankaccoun0\_.id as id1\_1\_,

bankaccoun0\_.owner as owner2\_1\_,

bankaccoun0\_.account as account3\_1\_,

bankaccoun0\_.bank\_name as bank\_nam4\_1\_,

bankaccoun0\_.swift as swift5\_1\_

from

BANK\_ACCOUNT bankaccoun0\_

Hibernate:

select

creditcard0\_.id as id1\_2\_,

creditcard0\_.owner as owner2\_2\_,

creditcard0\_.card\_number as card\_num3\_2\_,

creditcard0\_.exp\_month as exp\_mont4\_2\_,

creditcard0\_.exp\_year as exp\_year5\_2\_

from

CREDIT\_CARD creditcard0\_

Иными словами, для каждого конкретного подкласса Hibernate использует отдельный SELECT-запрос.  
  
Другой важной проблемой при использовании данной стратегии будет сложность рефакторинга. Изменение названия полей в суперклассе вызовет необходимость изменения названий во многих таблицах и потребует ручного переименования (инструменты большинства IDE не учитывают @AttributeOverride). В случае, если в вашей схеме не 2 таблицы, а 50, это чревато большими временными затратами.  
  
Этот подход возможно использовать только для верхушки иерархии классов, где:  
  
**а)** Полиморфизм не нужен (выборку для конкретного подкласса Hibernate будет выполнять в один запрос -> производительность будет высокой)  
  
**б)** Изменения в суперклассе не предвидятся.  
  
Для приложения, где запросы будут ссылаться на родительский класс BillingDetails эта стратегия не подойдет.

Стратегия 2

Одна таблица для каждого класса с объединениями (UNION)

В роли абстрактного класса вновь выступит BillingDetails.  
Схема БД также останется без почти без изменений.  
  
Единственный момент – поле CC\_OWNER в таблице CREDIT\_CARD придется переименовать в OWNER, поскольку данная стратегия не поддерживает @AttributeOverride. Из [документации](https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch10.html):  
*«The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables».*  
  
Новой также будет указанная над суперклассом аннотация @Inheritance с указанием выбранной стратегии TABLE\_PER\_CLASS.  
  
**ВАЖНО!** В рамках данной стратегии **наличие идентификатора в суперклассе является обязательным требованием** (в первом примере мы обошлись без него).  
  
**ВАЖНО!** Согласно стандарту JPA стратегия TABLE\_PER\_CLASS не является обязательной, поэтому другими реализациями может не поддерживаться.

Измененный Java-код

import javax.persistence.\*;

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

public abstract class BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

private String owner;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "CREDIT\_CARD")

public class CreditCard extends BillingDetails {

@Column(name = "card\_number")

private int cardNumber;

@Column(name = "exp\_month")

private String expMonth;

@Column (name = "exp\_year")

private String expYear;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "BANK\_ACCOUNT")

public class BankAccount extends BillingDetails {

private int account;

@Column(name = "bank\_name")

private String bankName;

private String swift;

Constructor, Getter, Setter, toString()

}

Наша схема SQL по-прежнему ничего не знает о наследовании; между таблицами нет никаких отношений.  
  
Главное преимущество данной стратегии можно увидеть, выполнив полиморфный запрос из предыдущего примера.

SELECT bd FROM BillingDetails bd

На сей раз он будет выполнен по-другому:  
  
Hibernate:

select

billingdet0\_.id as id1\_1\_,

billingdet0\_.owner as owner2\_1\_,

billingdet0\_.card\_number as card\_num1\_2\_,

billingdet0\_.exp\_month as exp\_mont2\_2\_,

billingdet0\_.exp\_year as exp\_year3\_2\_,

billingdet0\_.account as account1\_0\_,

billingdet0\_.bank\_name as bank\_nam2\_0\_,

billingdet0\_.swift as swift3\_0\_,

billingdet0\_.clazz\_ as clazz\_

from

( select

id,

owner,

card\_number,

exp\_month,

exp\_year,

null::int4 as account,

null::varchar as bank\_name,

null::varchar as swift,

1 as clazz\_

from

CREDIT\_CARD

union

all select

id,

owner,

null::int4 as card\_number,

null::varchar as exp\_month,

null::varchar as exp\_year,

account,

bank\_name,

swift,

2 as clazz\_

from

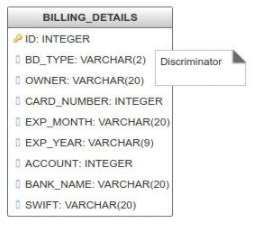
BANK\_ACCOUNT

) billingdet0\_

В данном случае Hibernate использует FROM, чтобы извлечь все экземпляры BillingDetails из всех таблиц подклассов. Таблицы объединяются с помощью UNION, а в промежуточный результат добавляются литералы (1 и 2). Литералы используются Hibernate для создания экземпляра правильного класса.  
  
Объединение таблиц требует одинаковой структуры столбцов, поэтому вместо несуществующих столбцов были вставлены NULL (например, «null::varchar as bank\_name» в credit\_card – в таблице кредиток нет названия банка).  
  
Другим важный преимуществом по сравнению с первой стратегией будет возможность использовать полиморфные ассоциации. Теперь можно будет без проблем отобразить ассоциации между классами User и BillingDetails.

Стратегия 3

Единая таблица для всей иерархии классов

Иерархию классов можно целиком отобрать в одну таблицу. Она будет содержать столбцы для всех полей каждого класса иерархии. Для каждой записи конкретный подкласс будет определяться значением дополнительного столбца с **селектором**.  
  
Наша схема теперь выглядит вот так:  


Запрос для создания

create table billing\_details

(

id serial not null

constraint billing\_details\_pkey

primary key,

bd\_type varchar(2),

owner varchar(20),

card\_number integer,

exp\_month varchar(9),

exp\_year varchar(4),

account integer,

bank\_name varchar(20),

swift varchar(20)

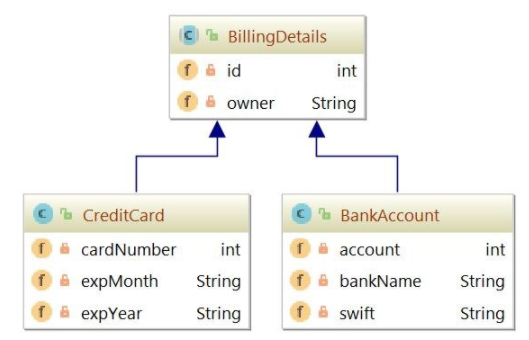
)

;

create unique index billing\_details\_card\_number\_uindex

on billing\_details (card\_number)

;

Структура Java-классов:  
  
  
  
Для создания отображения с одной таблицей необходимо использовать стратегию наследования SINGLE\_TABLE.  
Корневой класс будет отображен в таблицу BILLING\_DETAILS. Для различения типов будет использован столбец селектора. Он не является полем сущности и создан только для нужд Hibernate. Его значением будут строки – “CC” или “BA”.  
**ВАЖНО!**Если не указать столбец селектора в суперклассе явно – он получит название по умолчанию DTYPE и тип VARCHAR.  
  
Каждый класс иерархии может указать свое значение селектора с помощью аннотации @DiscriminatorValue.  
Не стоит пренебрегать явным указанием имени селектора: по умолчанию Hibernate будет использовать полное имя класса или имя сущности (зависит от того, используются ли файлы XML-Hibernate или xml-файлы JPA/аннотации).

Измененный Java-код

import javax.persistence.\*;

@Entity

@Table(name = "BILLING\_DETAILS")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "BD\_TYPE")

public abstract class BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

private String owner;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@DiscriminatorValue("BA")

public class BankAccount extends BillingDetails {

private int account;

@Column(name = "bank\_name")

private String bankName;

private String swift;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@DiscriminatorValue("CC")

public class CreditCard extends BillingDetails {

@Column(name = "card\_number")

private int cardNumber;

@Column(name = "exp\_month")

private String expMonth;

@Column (name = "exp\_year")

private String expYear;

Constructor, Getter, Setter, toString()

}

Для проверки используем в методе main уже привычный запрос

SELECT bd FROM BillingDetails bd

В случае с единой таблицей этот запрос будет выполнен так:  
  
Hibernate:

select

billingdet0\_.id as id2\_0\_,

billingdet0\_.owner as owner3\_0\_,

billingdet0\_.card\_number as card\_num4\_0\_,

billingdet0\_.exp\_month as exp\_mont5\_0\_,

billingdet0\_.exp\_year as exp\_year6\_0\_,

billingdet0\_.account as account7\_0\_,

billingdet0\_.bank\_name as bank\_nam8\_0\_,

billingdet0\_.swift as swift9\_0\_,

billingdet0\_.BD\_TYPE as BD\_TYPE1\_0\_

from

BILLING\_DETAILS billingdet0\_

Если же запрос выполняется к конкретному подклассу – будет просто добавлена строка «where BD\_TYPE = “CC”».  
  
Вот как будет выглядеть отображение в единую таблицу:  
  
  
В случае, когда схема была унаследована, и добавить в нее столбец селектора невозможно, на помощь приходит аннотация @DiscriminatorFormula, которую необходимо добавить к родительскому классу. В нее необходимо передать выражение CASE...WHEN.  
  
import org.hibernate.annotations.DiscriminatorFormula;

import javax.persistence.\*;

@Entity

@Table(name = "BILLING\_DETAILS")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorFormula("CASE WHEN CARD\_NUMBER IS NOT NULL THEN 'CC' ELSE 'BA' END")

public abstract class BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

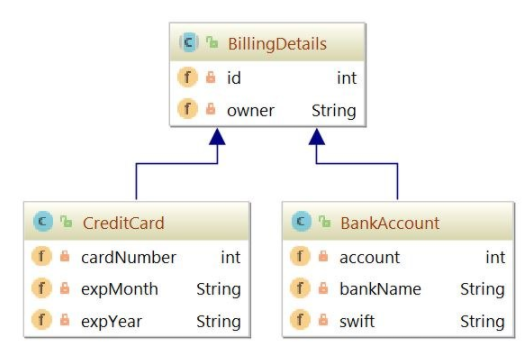
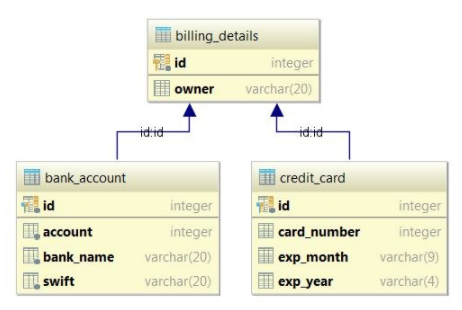
//.................

}

Главным плюсом данной стратегии является производительность. Запросы (как полиморфные, так и неполиморфные) выполняются очень быстро и могут быть легко написаны вручную. Не приходится использовать соединения и объединения. Эволюция схемы также производится очень просто.  
  
Однако, проблемы, сопровождающие эту стратегию, часто будут перевешивать ее преимущества.  
  
Главной из них является целостность данных. Столбцы тех свойств, которые объявлены в подклассах, могут содержать NULL. В результате простая программная ошибка может привести к тому, что в базе данных окажется кредитная карта без номера или без срока действия.  
  
Другой проблемой будет нарушение нормализации, а конкретно – [третьей нормальной формы](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B5%D1%82%D1%8C%D1%8F_%D0%BD%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%84%D0%BE%D1%80%D0%BC%D0%B0). В этом свете выгоды от повышенной производительности уже выглядят сомнительно. Ведь придется, как минимум, пожертвовать удобством сопровождения: в долгосрочной перспективе денормализованные схемы не сулят ничего хорошего.

Стратегия 4

Одна таблица для каждого класса с использованием соединений (JOIN)

Схема наших классов останется неизменной:  
  
  
  
А вот в схеме БД произошли некоторые изменения  
  


Запрос для создания BILLING\_DETAILS

create table billing\_details

(

id integer not null

constraint billing\_details\_pkey

primary key,

owner varchar(20) not null

)

;

Для CREDIT\_CARD

create table credit\_card

(

id integer not null

constraint credit\_card\_pkey

primary key

constraint credit\_card\_billing\_details\_id\_fk

references billing\_details,

card\_number integer not null,

exp\_month varchar(255) not null,

exp\_year varchar(255) not null

)

;

create unique index credit\_card\_card\_number\_uindex

on credit\_card (card\_number)

;

Для BANK\_ACCOUNT

create table bank\_account

(

id integer not null

constraint bank\_account\_pkey

primary key

constraint bank\_account\_billing\_details\_id\_fk

references billing\_details,

account integer not null,

bank\_name varchar(255) not null,

swift varchar(255) not null

)

;

create unique index bank\_account\_account\_uindex

on bank\_account (account)

;

В Java-коде для создания такого отображения необходимо использовать стратегию JOINED.

Измененный Java-код

import javax.persistence.\*;

@Entity

@Table(name = "BILLING\_DETAILS")

@Inheritance(strategy = InheritanceType.JOINED)

public abstract class BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

private String owner;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "CREDIT\_CARD")

public class CreditCard extends BillingDetails {

@Column(name = "card\_number")

private int cardNumber;

@Column(name = "exp\_month")

private String expMonth;

@Column (name = "exp\_year")

private String expYear;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@Table(name = "BANK\_ACCOUNT")

public class BankAccount extends BillingDetails {

private int account;

@Column(name = "bank\_name")

private String bankName;

private String swift;

Constructor, Getter, Setter, toString()

}

Теперь при сохранении, например, экземпляра CreditCard Hibernate вставит две записи. В таблицу BILLING\_DETAILS попадут свойства, объявленные в полях суперкласса BillingDetails, а значения полей подкласса CreaditCard будут записаны в таблицу CREDIT\_CARD. Эти записи будут объединены общим первичным ключом.  
  
Таким образом, схема была приведена в нормальное состояние. Эволюция схемы и определение ограничений целостности также осуществляются просто.  
Внешние ключи позволяют представить полиморфную ассоциацию с конкретным подклассом.  
  
Выполнив запрос

SELECT bd FROM BillingDetails bd

, мы увидим следующую картину:  
  
Hibernate:

select

billingdet0\_.id as id1\_1\_,

billingdet0\_.owner as owner2\_1\_,

billingdet0\_1\_.card\_number as card\_num1\_2\_,

billingdet0\_1\_.exp\_month as exp\_mont2\_2\_,

billingdet0\_1\_.exp\_year as exp\_year3\_2\_,

billingdet0\_2\_.account as account1\_0\_,

billingdet0\_2\_.bank\_name as bank\_nam2\_0\_,

billingdet0\_2\_.swift as swift3\_0\_,

case

when billingdet0\_1\_.id is not null then 1

when billingdet0\_2\_.id is not null then 2

when billingdet0\_.id is not null then 0

end as clazz\_

from

BILLING\_DETAILS billingdet0\_

left outer join

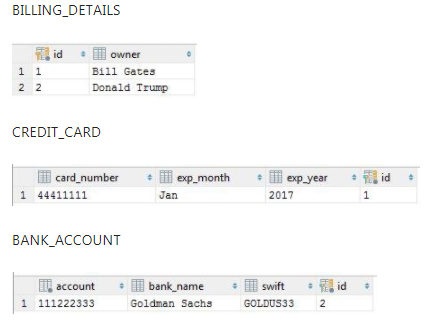
CREDIT\_CARD billingdet0\_1\_

on billingdet0\_.id=billingdet0\_1\_.id

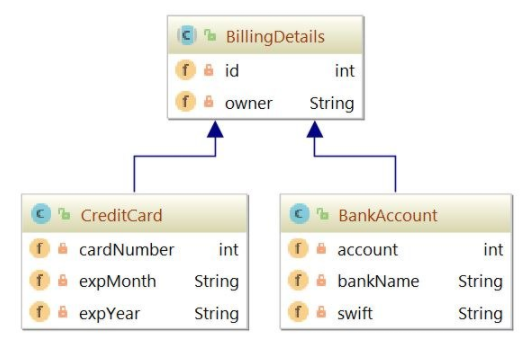
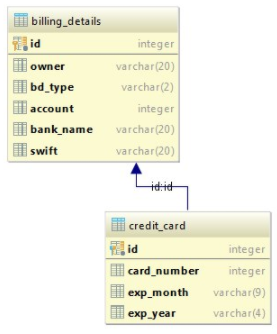
left outer join

BANK\_ACCOUNT billingdet0\_2\_

on billingdet0\_.id=billingdet0\_2\_.id

  
Предложение CASE…WHEN позволяет Hibernate определить конкретный подкласс для каждой записи. В нем проверяется наличие либо отсутствие строк в таблицах подклассов CREDIR\_CARD и BANK\_ACCOUNT с помощью литералов.  
  
Подобную стратегию будет весьма непросто реализовать вручную. Даже реализовать отчеты на основе произвольных запросов будет значительно сложнее.  
Производительность также может оказаться неприемлемой для конкретного проекта, поскольку запросы потребуют соединения нескольких таблиц или многих последовательных операций чтения.

Смешение стратегий отображения наследования

При работе со стратегиями TABLE\_PER\_CLASS, SINGLE\_TABLE и JOINED значительным неудобством является тот факт, что между ними невозможно переключаться. Выбранной стратегии придется придерживаться до конца (либо полностью менять схему).  
Но есть приемы, с помощью которых можно переключить стратегию отображения для конкретного подкласса.  
  
Например, отобразив иерархию классов в единственную таблицу (стратегия 3), можно выбрать для отдельного подкласса стратегию с отдельной таблицей и внешним ключом (стратегия 4).  
  
  


Скрипт для создания BILLING\_DETAILS

create table billing\_details

(

id integer not null

constraint billing\_details\_pkey

primary key,

owner varchar(20),

account integer,

bank\_name varchar(20),

swift varchar(20)

)

;

Для CREDIT\_CARD

create table credit\_card

(

card\_number integer not null,

exp\_month varchar(255) not null,

exp\_year varchar(255) not null,

id integer not null

constraint credit\_card\_pkey

primary key

constraint fksf645frtr6h3i4d179ff4ke9h

references billing\_details

)

;

Теперь мы можем отобразить подкласс CreditCard в отдельную таблицу.  
Для этого нам нужно будет применить стратегию InheritanceType.SINGLE\_TABLE к суперклассу BillingDetails, а в работе с классом CreditCard нам поможет аннотация @SecondaryTable.

Измененный Java-код

import javax.persistence.\*;

@Entity

@Table(name = "BILLING\_DETAILS")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "BD\_TYPE")

public abstract class BillingDetails {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE)

private int id;

private String owner;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

public class BankAccount extends BillingDetails {

private int account;

@Column(name = "bank\_name")

private String bankName;

private String swift;

Constructor, Getter, Setter, toString()

}

import javax.persistence.\*;

@Entity

@DiscriminatorValue("CC")

@SecondaryTable(name = "CREDIT\_CARD",

pkJoinColumns = @PrimaryKeyJoinColumn(name = "ID"))

public class CreditCard extends BillingDetails {

@Column(table = "CREDIT\_CARD",name = "card\_number")

private int cardNumber;

@Column(table = "CREDIT\_CARD",name = "exp\_month")

private String expMonth;

@Column (table = "CREDIT\_CARD",name = "exp\_year")

private String expYear;

public CreditCard() {

}

Getter, Setter, toString()

}

При помощи аннотаций @SecondaryTable и @Column мы переопределяем основную таблицу и ее столбцы, указывая Hibernate, откуда необходимо брать данные.  
  
При выборе стратегии SINGLE\_TABLE столбцы подклассов могут содержать NULL. Используя же данный прием, вы можете гарантировать целостность данных для конкретного подкласса (в нашем случае — CreditCard).  
Исполняя полиморфный запрос, Hibernate выполнит внешнее соединение для извлечения экземпляров BillingDetails и всех его подклассов.  
  
Давайте попробуем:

SELECT bd FROM BillingDetails bd

Результат:  
  
Hibernate:

select

billingdet0\_.id as id2\_0\_,

billingdet0\_.owner as owner3\_0\_,

billingdet0\_.account as account4\_0\_,

billingdet0\_.bank\_name as bank\_nam5\_0\_,

billingdet0\_.swift as swift6\_0\_,

billingdet0\_1\_.card\_number as card\_num1\_1\_,

billingdet0\_1\_.exp\_month as exp\_mont2\_1\_,

billingdet0\_1\_.exp\_year as exp\_year3\_1\_,

billingdet0\_.BD\_TYPE as BD\_TYPE1\_0\_

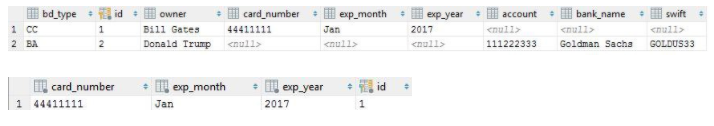
from

BILLING\_DETAILS billingdet0\_

left outer join

CREDIT\_CARD billingdet0\_1\_

on billingdet0\_.id=billingdet0\_1\_.ID

  
  
Этот прием можно применить и к остальным классам иерархии, но для обширной иерархии он подойдет не слишком хорошо, поскольку внешнее соединение в таком случае станет проблемой. Для такой иерархии лучше подойдет стратегия, которая немедленно выполнит второй SQL-запрос вместо внешнего соединения.

Выбор стратегии

Каждая из перечисленных выше стратегий и приемов имеет свои преимущества и недостатки. Общие рекомендации по выбору конкретной стратегии будут выглядеть так:

— **Стратегию №1** возможно использовать только для верхушки иерархии классов, где:  
а) Полиморфизм не нужен (выборку для конкретного подкласса Hibernate будет выполнять в один запрос -> производительность будет высокой)  
б) Изменения в суперклассе не предвидятся.  
Для приложения, где запросы будут ссылаться на родительский класс BillingDetails эта стратегия не подойдет.  
  
— **Стратегию №2** (**TABLE\_PER\_CLASS** на основе UNION), если полиморфные запросы и ассоциации не требуются. Если вы редко выполняете (или не выполняете вообще) **«select bd from BillingDetails bd»,** и у вас нет классов, ссылающихся на BillingDetails, этот вариант будет лучшим (поскольку возможность добавления оптимизированных полиморфных запросов и ассоциаций сохранится).  
  
— **Стратегию №3** (**SINGLE\_TABLE**) стоит использовать:  
а) Только для простых задач. В ситуациях, когда нормализация и ограничение NOT NULL являются критическими – следует отдать предпочтение стратегии №4 (JOINED). Имеет смысл задуматься, не стоит ли в данном случае вообще отказаться от наследования и [заменить его делегированием](https://refactoring.guru/ru/replace-inheritance-with-delegation)  
б) Если требуются полиморфные запросы и ассоциации, а также динамическое определение конкретного класса во время выполнения; при этом подклассы объявляют относительно мало новых полей и основная разница с суперклассом заключается в поведении.  
Ну и вдобавок к этому, Вам предстоит серьезный разговор с администратором БД.  
  
— **Стратегия №4** (**JOINED**) подойдет в случаях, когда требуются полиморфные запросы и ассоциации, но подклассы объявляют относительно много новых полей.  
  
!!Выбор между JOINED и TABLE\_PER\_CLASS требует оценки планов выполнения запросов на реальных данных, поскольку ширина и глубина иерархии наследования могут сделать стоимость соединений (и, как следствие, производительность) неприемлемыми!!  
  
Отдельно стоит принять во внимание, что аннотации наследования невозможно применить к интерфейсам.

Комментарии 12

8 сентября 2017 в 16:38

**Наследование на уровне SQL красиво и ИМХО правильно реализуется примерно так же, как оно физически реализовано в объектно-ориентированных языках программирования. Класс-родитель представляется в виде первой таблицы, класс-потомок представляется в виде второй таблицы с отношением @OneToOne к первой. Унаследованные поля/свойства хранит в первой таблице, новые — во второй.**

**[jd2050](https://habr.com/ru/users/jd2050/)**8 сентября 2017 в 16:53

Добрый день!  
В принципе, Вы правы. Об этом как раз говорится здесь:

| Стратегия №4 (JOINED) подойдет в случаях, когда требуются полиморфные запросы и

| ассоциации, но подклассы объявляют относительно много новых полей.

Другое дело, что если их нет, то нет и особого смысла выбирать «заточенную» под них стратегию JOINED, при наличии более выигрышных в плане производительности вариантов.

8 сентября 2017 в 19:11

Странно, почему нет смешанной стратегии JOINED и SINGLE\_TABLE. Родительская таблица содержит поля первичного ключа, дискриминатор и поля нагрузки. Дочерние таблицы — поля первичного ключа и поля своей нагрузки.

Как я понимаю, основной недостаток JOINED — когда у нас десятки классов в иерархии, соединяться будут все десятки таблиц только ради того, чтобы выяснить для каждой строки, откуда брать данные.

**[DrPass](https://habr.com/ru/users/DrPass/)**9 сентября 2017 в 03:35

| Как я понимаю, основной недостаток JOINED — когда и нас десятки классов в иерархии

Так и есть, но модель данных обычно не имеет зверских иерархий, там как правило один (а чаще, пожалуй, вообще ноль), ну максимум два уровня наследования.

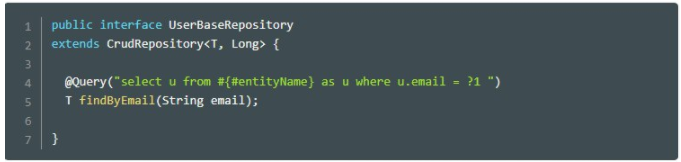
9 сентября 2017 в 13:46

Ну количество таблиц-то зависит не только от глубины дерева наследования, но и его ширины. У вас может быть один суперкласс и десятки его прямых потомков. Мне кажется, вполне не редкая ситуация, особенно, если в этом месте система предполагает расширяемость (добавление новых классов). И при каждой выборке объектов суперкласса придется джойнить все эти таблицы. Надо надеяться, в Hibernate есть оптимизации, когда он не джойнит заведомо ненужные "братские" таблицы, если выбираем какого-то потомка.

8 сентября 2017 в 19:23

Мне лично не понятны не стратегии, а как дальше работать с наследуемыми entity.  
Например, как определить Spring'овый JPA Repository для BillingDetails в данном случае?  
  
Или как определить класс Payment со связью к BankAccount или CreditCard?

11 сентября 2017 в 21:19

Добрый вечер!  
Например, вот так:  
   
  
Пример не связан с исходниками статьи, но суть понятна. Используя SpEL, указываете в [Query](https://habrahabr.ru/users/query/) нужную реализацию User'a, которая и будет выступать в качестве T.

8 сентября 2017 в 21:12 **+1**

Наследование в SQL бывает — см. <https://www.postgresql.org/docs/current/static/ddl-inherit.html>. А еще как бы «предпочитайте композицию наследованию» :)

11 сентября 2017 в 21:12

Добрый вечер!  
Бывает, но только в качестве фич конкретных СУБД:) На мой взгляд, при реализации лучше все-таки держать в голове вероятность смены провайдера.

9 сентября 2017 в 03:43 **+2**

Как насчет самого простого варианта?  
"0. Не используйте наследование реализаций для объектов, отображаемых на базу данных."

9 сентября 2017 в 11:15

Ну, грубо первый вариант примерно об этом же. Мне тоже кажется это самым нормальным: сделать отдельные таблицы и общий интерфейс для соответствующих объектов. Но, как было подмечено, изменение родительского класса (интерфейса) затронет всех потомков. Это может вылиться во множество изменений.

9 сентября 2017 в 14:10

А не надо менять опубликованные интерфейсы, это и без базы данных слишком дорого.