1.1

1.2 持久化类的编写规则

1.2.1 持久化类的概述

- 持久化:将内存中的一个对象持久化到数据库中的过程.Hibernate 框架就是用来持久化的框架
- 持久化类:一个 Java 类与数据库中的表建立映射关系.那么这个类在 Hibernate 中称为持久化类
    - 持久化类 = Java 类 + 映射文件

1.2.2 持久化类的编写规则

- 对持久化类提供一个无参构造方法 :Hibernate 底层需要使用反射生成实例
- 属性需要私有,私有的属性提供 get 和 set 方法 :Hibernate 中获取、设置对象的值
- 对持久化类提供一个唯一标识 OID 与数据库主键对应 :Java 中通过内存地址区分是否是同一个对象,数据库中通过主键区分是否为同一条记录,Hibernate 中通过 OID 区分是否为同一个对象
- 持久化类中属性尽量使用包装类 :因为基本数据类型默认值是:0,0有很多歧义.包装类默认值为 null
- 持久化类不要使用 final 修饰 :延迟加载本身是 Hibernate 的一个优化手段,返回一个代理对象(javassist可以对没有实现接口的类产生代理--使用类非常底层的字节码增强技术,继承这个类进行代理)。如果不能被继承，就不能产生代理对象，延迟加载也就失效了。load 和 get 方法一致。

1.3 主键生成策略

1.3.1 主键的分类

1.3.1.1 自然主键

- 自然主键:主键本身就是表中的一个字段(实体类中的一个属性)

> 创建一个人员表,使用身份证(唯一不可重复)作为主键,这种主键称为自然主键

1.3.1.2 代理主键

- 代理主键:主键本身不是表中必须的一个字段(不是实体中的具体的属性)

> 创建一个人员表,不使用身份证作为主键,使用不相关的ID作为主键,这种主键称为代理主键

- 实际开发中,尽量使用代理主键

1.3.2 主键的生成策略

1.3.2.1 Hibernate 的主键生成策略

在实际开发中一般不允许用户手动设置主键，一般将主键交给数据库，或者手动编写程序进行设置。在 Hibernate 中为了减少代码编写，提供了很多种主键生成的策略。

- increment

  > Hibernate 中提供的自动增长机制,适用 short int lang 类型的主键,在单线程程序中使用
  >
  > 首先发送一条SQL语句:select max(id) from 表; 然后让 id+1,作为下一条记录的主键

- identity

  > 适用 short int lang 类型的主键,使用的是数据库底层的自动增长机制.适用于有自动增长机制的数据库(MySQL), Oracle 没有自动增长机制.

- sequence

  > 适用于 short int lang 类型的主键,采用序列的方式.(Oracle支持序列),MySQL 不支持序列.

- uuid

  > 适用于字符串类型的主键,使用 Hibernate 中随机的方式生成字符串主键

- native

  > 本地策略,可以在 identity 和 sequence 之间切换.

- assigned

  > Hibernate 放弃外键的管理,需要通过手动编写程序和用户自己设置.

- foreign

  > 外部的,在一对一的关联映射情况下使用.

1.4 持久化类的三种状态

Hibernate 是持久层的 ORM 框架，专注于数据的持久化工作。在进行数据持久化操作时，持久化对象可能处于不同的状态当中。这些状态可分为三种，分别为瞬时态、持久态和脱管态。下面分别针对这三种状态进行简单介绍。

1）瞬时态（transient）

瞬时态也称为临时态或者自由态，瞬时态的对象是由 new 关键字开辟内存空间的对象，不存在持久化标识 OID（相当于主键值），且未与任何的 Session 实例相关联，在数据库中也没有记录，失去引用后将被 JVM 回收。瞬时对象在内存孤立存在，它是携带信息的载体，不和数据库的数据有任何关联关系。

2）持久态（persistent）

持久态的对象存在一个持久化标识 OID，当对象加入到 Session 缓存中时，就与 Session 实例相关联。它在数据库中存在与之对应的记录，每条记录只对应唯一的持久化对象。需要注意的是，持久态对象是在事务还未提交前变成持久态的。

3）脱管态（detached）

脱管态也称离线态或者游离态，当持久化对象与 Session 断开时就变成了脱管态，但是脱管态依然存在持久化标识 OID，只是失去了与当前 Session 的关联。需要注意的是，脱管态对象发生改变时 Hibernate 是不能检测到的。

1.4.2 三种状态的区分

```
package com.admiral.pojo;

import java.io.Serializable;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.utils.HibernateUtils;

public class HibernateDemo2 {

    @Test
    public void test1() {
        Session session = HibernateUtils.openSession();
        Transaction transaction = session.beginTransaction();

        Customer customer = new Customer();//瞬时态对象:没有唯一标识OID,没有被
session 管理
        customer.setCust_name("张三丰");

        Serializable id = session.save(customer);//持久态对象:有唯一表示 OID,被
session 管理

        Customer customer2 = session.get(Customer.class, id);

        transaction.commit();
        session.close();

        System.out.println("客户名称" + customer.getCust_name());//托管态对象:有唯一
表示 OID,没有被 session 管理.
    }

}
```

1.4.3 三种状态转换

1.4.3.1 三种状态转换图

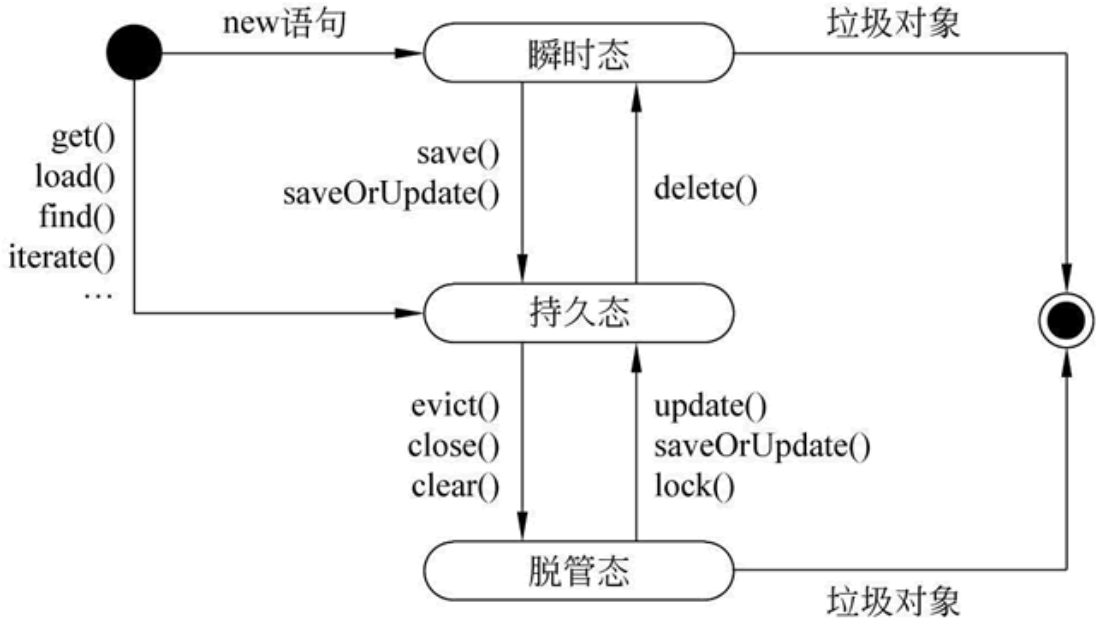


图 1 持久化对象的状态转换

1.4.3.2 瞬时态对象

- 获得
  - Customer customer = new Customer();
- 状态转换
  - 瞬时 --> 持久
    - save(Object obj)、saveOrUpdate(Object obj)
  - 瞬时 --> 托管
    - customer.setCust_id(1L);

1.4.3.3 持久态对象

- 获得
  - get()、load()
- 状态转换
  - 持久 --> 瞬时
    - delete();
  - 持久 --> 托管
    - close()、clear()、evict()

1.4.3.4 托管态对象

- 获得
  - Customer customer = new Customer();
  - customer.setCust_id(1L);
- 状态转换
  - 托管 --> 持久
    - update()、saveOrUpdate();
  - 托管 --> 瞬时
    - customer.setCust_id(null);

1.4.3.5 持久态对象的特性

1.4.3.5.1 持久态对象可以自动更新数据库

```
    @Test
    //持久态对象自动更新数据库
    public void test2() {
        Session session = HibernateUtils.openSession();
        Transaction transaction = session.beginTransaction();

        //获得持久态对象 get  load
        Customer customer = session.get(Customer.class, 1L);
        customer.setCust_name("欧阳锋");
//      session.update(customer);


        transaction.commit();
        session.close();

    }
```

1.5 Hibernate 的一级缓存

1.5.1 缓存的概述

1.5.2 Hibernate 的缓存

1.5.2.1 Hibernate 的一级缓存

Hibernate 的一级缓存就是指 Session 缓存，Session 缓存是一块内存空间，用来存放相互管理的 Java 对象，在使用 Hibernate 查询对象的时候，首先会使用对象属性的 OID 值在 Hibernate 的一级缓存中进行查找，如果找到匹配的 OID 值的对象，就直接将该对象从一级缓存中取出使用，不会再查询数据库，如果没有找到相同 OID 值的对象，则会去数据库中查找响应数据。当从数据库中查询到所需数据时，该数据信息也会放置到一级缓存中。Hibernate 的一级缓存的作用就是减少对数据库访问的次数。

在 Session 接口的实现中包含一系列的 Java 集合，这些 Java 集合构成了 Session 缓存。只要 Session 实例没有结束生命周期，存放在它缓存中的对象也不会结束生命周期。固一级缓存也被称为 Session 级别的缓存。

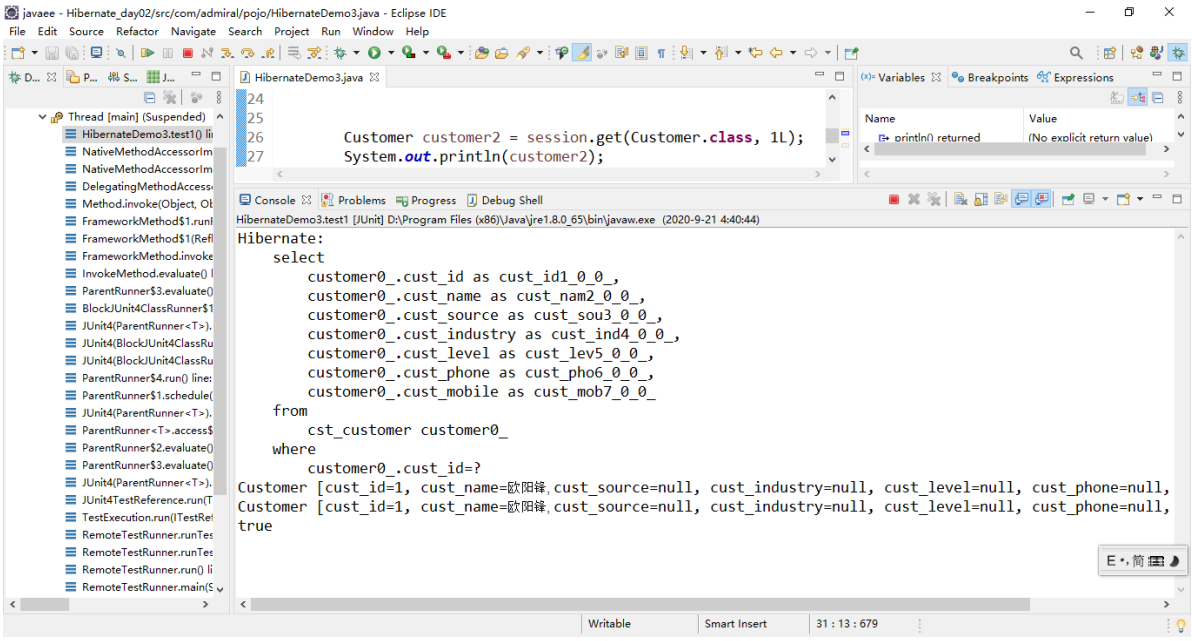
Hibernate 的一级缓存有如下特点：

- 当应用程序调用 Session 接口的 sava()、update、saveOrUpdate

1.5.2.2 证明 Hibernate 的一级缓存存在

```
package com.admiral.pojo;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.utils.HibernateUtils;

/**
 * Hibernate 的一级缓存测试
 * @author Administrator
 *
 */
public class HibernateDemo3 {

    @Test
    //证明Hibernate 的一级缓存存在
    public void test1() {
        Session session = HibernateUtils.openSession();
        Transaction transaction = session.beginTransaction();

        Customer customer1 = session.get(Customer.class, 1L);
        System.out.println(customer1);


        Customer customer2 = session.get(Customer.class, 1L);
        System.out.println(customer2);
```

```
        System.out.println(customer1 == customer2);

        transaction.commit();
        session.close();
    }
}
```



1.5.2.3 一级缓存快照区

1.6 Hibernate 事务管理

1.6.1 绑定线程的 Session

Hibernate5 中自身提供了三种管理 Session 对象的方法

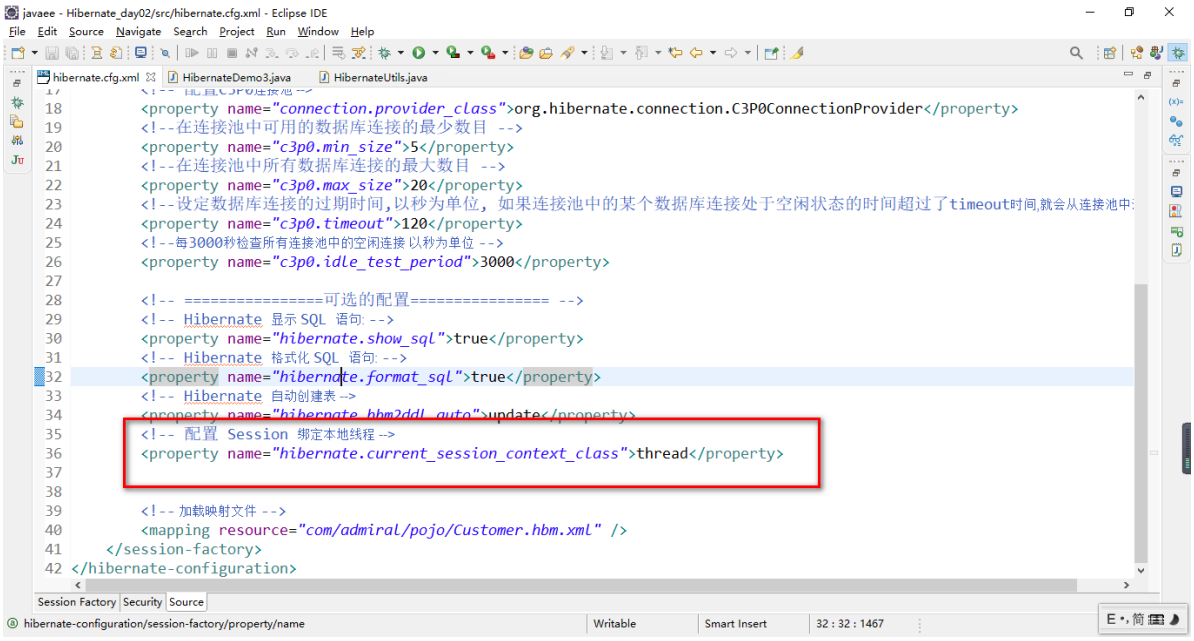
- Session 对象的生命周期与本地线程绑定
- Session 对象的生命周期与 JTA 事务绑定
- Hibernate 委托程序管理 Session 对象的声明周期

在 Hibernate 配置文件中,hibernate.current_session_context_class 属性用于指定 Session 管理方式,可选值包括

- thread: Session 对象的生命周期与本地线程绑定
- jta: Session 对象的生命周期与 JTA 事务绑定
- managed: Hibernate 委托程序管理 Session 对象的声明周期

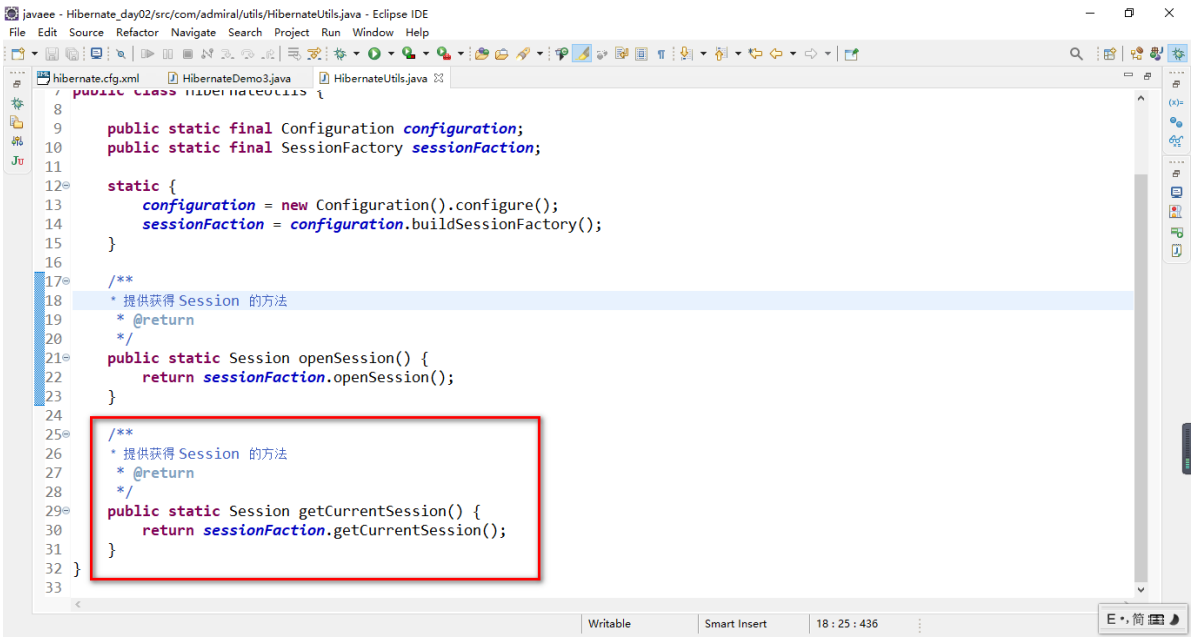
在 hibernate.cfg.xml 配置文件中进行如下配置:

```
        <!-- 配置 Session 绑定本地线程 -->
        <property
name="hibernate.current_session_context_class">thread</property>
```



修改 HibernateUtils 工具类

```
    /**
     *  提供获得 Session 的方法
     * @return
     */
    public static Session getCurrentSession() {
        return sessionFaction.getCurrentSession();
    }
```



测试

```java
@Test
// 测试 Session 绑定本地线程
public void test3() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Customer customer = new Customer();
    customer.setCust_name("洪七公");
    session.save(customer);

    transaction.commit();
}
```

1.6 Hibernate 的其他 API

1.6.1 Query

Query 代表面向对象的一个 Hibernate 操作. 在 Hibernate 中通常使用 session.createQuery() 方法接收一个 HQL 语句,然后使用 query 的 list() 或 uniqueResult() 方法执行查询. 所谓的 HQL 是 Hibernate Query Language 的缩写,其语法很像 SQL 但是它是完全面向对象的.

在 Hibernate 中使用 Query 的步骤

1. 获得 Hibernate 的 Session 对象
2. 编写 HQL 语句
3. 调用 session.createQuery() 创建查询对象
4. 如果 HQL 语句包含参数,则调用 Query 的 setXXX 方法设置参数
5. 调用 Query 的 list() 或 uniqueResult() 方法执行查询

查询所有

```java
package com.admiral.pojo;

import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
```

```
import org.junit.Test;

import com.admiral.utils.HibernateUtils;

public class HibernateDemo4 {

    @Test
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        String hql = "from Customer";
        Query query = session.createQuery(hql);
        List<Customer> customers = query.list();
        for (Customer customer : customers) {
            System.out.println(customer);
        }
        transaction.commit();
    }
}
```

条件查询

```
    @Test
    public void test2() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        String hql = "from Customer where cust_name like ?";
        Query query = session.createQuery(hql);
        query.setParameter(0, "小%");
        List<Customer> customers = query.list();
        for (Customer customer : customers) {
            System.out.println(customer);
        }


        transaction.commit();

    }
```

分页查询

```
    @Test
    public void test3() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        String hql = "from Customer";
        Query query = session.createQuery(hql);

        //设置分页
```

```
        query.setFirstResult(3);
        query.setMaxResults(3);
        List<Customer> customers = query.list();
        for (Customer customer : customers) {
            System.out.println(customer);
        }


        transaction.commit();

    }
```

1.6.2 Criteria

Criteria:QBC查询(Query By Criteria)

> 更加面向对象的一种查询方式

- 查询所有

```
package com.admiral.pojo;

import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.utils.HibernateUtils;

public class HibernateDemo5 {

    @Test
    // Criteria
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        //通过 Session 获取 criteria
        Criteria criteria = session.createCriteria(Customer.class);
        List<Customer> list = criteria.list();
        for (Customer customer : list) {
            System.out.println(customer);
        }

        transaction.commit();
    }
}
```

- 条件查询

```
    @Test
    // Criteria
```

```
    public void test2() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        //通过 Session 获取 criteria
        Criteria criteria = session.createCriteria(Customer.class);

        //设置条件
        criteria.add(Restrictions.ilike("cust_name", "小%"));

        List<Customer> list = criteria.list();
        for (Customer customer : list) {
            System.out.println(customer);
        }

        transaction.commit();
    }
```

- 分页查询

```
    @Test
    // Criteria
    public void test3() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        //通过 Session 获取 criteria
        Criteria criteria = session.createCriteria(Customer.class);

        //设置分页
        criteria.setFirstResult(0);
        criteria.setMaxResults(3);
        List<Customer> list = criteria.list();
        for (Customer customer : list) {
            System.out.println(customer);
        }

        transaction.commit();
    }
```

1.6.3 SQLQuery