# 双向一多

## 概念

双向一对多关系中，必须存在一个关系维护端，在 JPA 规范中，要求 many 的一方作为关系的维护端(owner side), one 的一方作为被维护端(inverse side)。

可以在 one 方指定 @OneToMany 注释并设置 mappedBy 属性，以指定它是这一关联中的被维护端，many 为维护端。

## 配置

一个订单只有一个用户，一个用户多个订单

使用 @ManyToOne 来映射多对一的关联关系， fetch 属性来修改默认的关联属性的加载策略

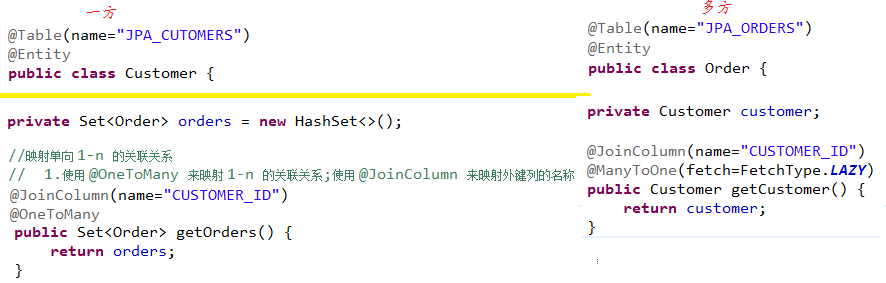
使用 @JoinColumn 来映射外键.

@JoinColumn(name="CUSTOMER\_ID")

@OneToMany

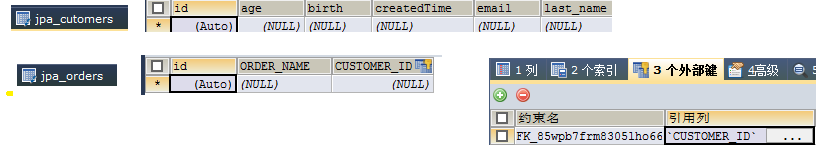
@JoinColumn(name="CUSTOMER\_ID")

@ManyToOne(fetch=FetchType.***LAZY***)



## 数据生成信息

Order表会产生外键



## 测试准备

### Customer

@Table(name="JPA\_CUTOMERS")

@Entity

**public** **class** Customer {

**private** Integer id;

**private** String lastName;

**private** String email;

**private** **int** age;

**private** Date createdTime;

**private** Date birth;

**private** Set<Order> orders = **new** HashSet<>();

@GeneratedValue(strategy=GenerationType.***AUTO***)

@Id

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

@Column(name = "last\_name", length = 50, nullable = **false**)

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Temporal(TemporalType.***TIMESTAMP***)

**public** Date getCreatedTime() {

**return** createdTime;

}

**public** **void** setCreatedTime(Date createdTime) {

**this**.createdTime = createdTime;

}

@Temporal(TemporalType.***DATE***)

**public** Date getBirth() {

**return** birth;

}

**public** **void** setBirth(Date birth) {

**this**.birth = birth;

}

//映射单向 1-n 的关联关系

// 1.使用 @OneToMany 来映射 1-n 的关联关系;使用 @JoinColumn 来映射外键列的名称

// 2.可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略

// 3.可以通过 @OneToMany 的 cascade 属性来修改默认的删除策略.

// @JoinColumn(name="CUSTOMER\_ID")

// @OneToMany

// @OneToMany(fetch=FetchType.EAGER)

// @OneToMany(fetch=FetchType.EAGER,cascade={CascadeType.REMOVE})

@OneToMany(fetch=FetchType.***EAGER***,cascade={CascadeType.***REMOVE***},mappedBy="customer")

**public** Set<Order> getOrders() {

**return** orders;

}

**public** **void** setOrders(Set<Order> orders) {

**this**.orders = orders;

}

}

### Order

@Table(name="JPA\_ORDERS")

@Entity

**public** **class** Order {

**private** Integer id;

**private** String orderName;

@GeneratedValue

@Id

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

@Column(name="ORDER\_NAME")

**public** String getOrderName() {

**return** orderName;

}

**public** **void** setOrderName(String orderName) {

**this**.orderName = orderName;

}

}

### 准备JPATest

**public** **class** JPATest {

**private** EntityManagerFactory entityManagerFactory;

**private** EntityManager entityManager;

**private** EntityTransaction transaction;

@Before

**public** **void** init(){

entityManagerFactory = Persistence.*createEntityManagerFactory*("jpa\_test1");

entityManager = entityManagerFactory.createEntityManager();

transaction = entityManager.getTransaction();

transaction.begin();

}

@After

**public** **void** destroy(){

transaction.commit();

entityManager.close();

entityManagerFactory.close();

}

}

# 操作

## 保存（双向维护）

//若是双向 1-n 的关联关系, 执行保存时

//1.若先保存 n 的一端, 再保存 1 的一端, 默认情况下, 会多出 n 条 UPDATE 语句.

//2.若先保存 1 的一端,再保存 n 的一端, 则会多出 n\*2 条 UPDATE 语句

//3.在进行双向 1-n 关联关系时, 建议使用 n 的一方来维护关联关系, 而 1 的一方不维护关联系, 这样会有效的减少 SQL 语句.

//注意: 若在 1 的一端的 @OneToMany 中使用 mappedBy 属性, 则 @OneToMany 端就不能再使用 @JoinColumn 属性了.

### 先一再n

//若是双向 1-n 的关联关系, 执行保存时

//1.若先保存 n 的一端, 再保存 1 的一端, 默认情况下, 会多出 n 条 UPDATE 语句.

// 1.1 先插入customer，得到id，此时customer\_id有值，插入order

// 1.2 只需要维护多方外键关联关系，

@Test

**public** **void** testOneToManyPersist1(){//先n再一

Customer customer = **new** Customer();

customer.setAge(18);

customer.setBirth(**new** Date());

customer.setCreatedTime(**new** Date());

customer.setEmail("mm@163.com");

customer.setLastName("MM");

Order order1 = **new** Order();

order1.setOrderName("O-MM-1");

Order order2 = **new** Order();

order2.setOrderName("O-MM-2");

//建立关联关系

customer.getOrders().add(order1);

customer.getOrders().add(order2);

order1.setCustomer(customer);

order2.setCustomer(customer);

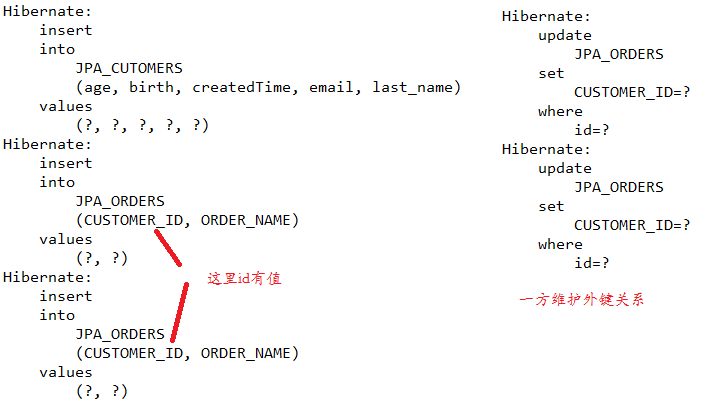
//执行保存操作

entityManager.persist(customer);

entityManager.persist(order1);

entityManager.persist(order2);

}



### 先n再一

//2.若先保存 1 的一端,再保存 n 的一端, 则会多出 n方\*2 条 UPDATE 语句

// 2.1 先插入order，此时customer\_id为空，再插入customer，得到id

// 2.2先维护多方外键关联关系，

// 2.3由于双向维护，再维护一方外键关系，实际是不需要的

@Test

**public** **void** testOneToManyPersist2(){//先n再一

Customer customer = **new** Customer();

customer.setAge(18);

customer.setBirth(**new** Date());

customer.setCreatedTime(**new** Date());

customer.setEmail("mm@163.com");

customer.setLastName("MM");

Order order1 = **new** Order();

order1.setOrderName("O-MM-1");

Order order2 = **new** Order();

order2.setOrderName("O-MM-2");

//建立关联关系

customer.getOrders().add(order1);

customer.getOrders().add(order2);

order1.setCustomer(customer);

order2.setCustomer(customer);

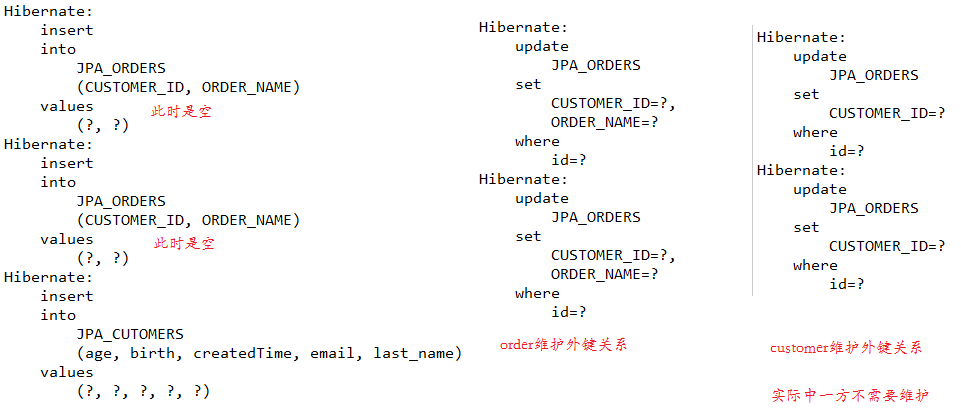
//执行保存操作

entityManager.persist(order1);

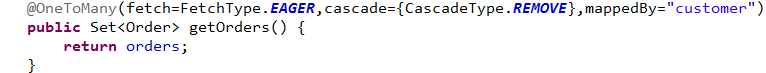
entityManager.persist(order2);

entityManager.persist(customer);

}



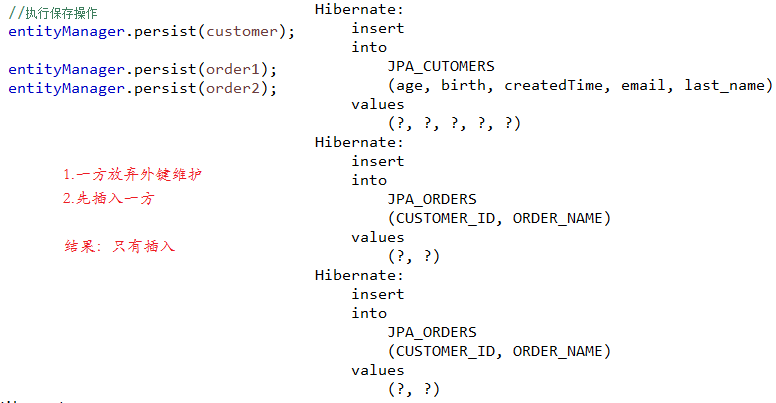
### 一方放弃维护，只维护多方（推荐）



//3.在进行双向 1-n 关联关系时, 建议使用 n 的一方来维护关联关系, 而 1 的一方不维护关联系, 这样会有效的减少 SQL 语句.

// 注意: 若在 1 的一端的 @OneToMany 中使用 mappedBy 属性, 则 @OneToMany 端就不能再使用 @JoinColumn 属性了.

// @OneToMany(fetch=FetchType.EAGER,cascade={CascadeType.REMOVE},mappedBy="customer")



## 查询（oid）

### 默认懒加载

查询一方；懒加载：当调用时，才发送查询sql语句

//默认对关联的多的一方使用懒加载的加载策略.

//可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略

@Test

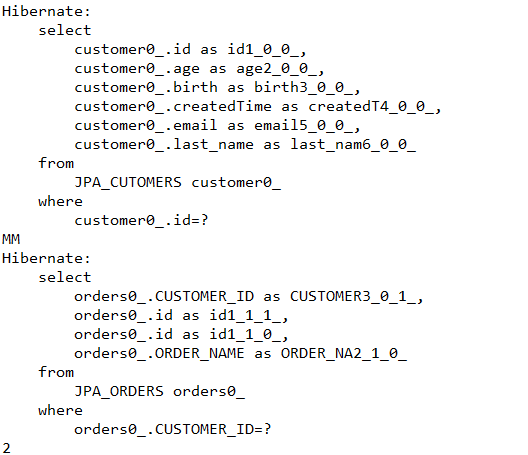
**public** **void** testOneToManyFind(){

Customer customer = entityManager.find(Customer.**class**, 3);

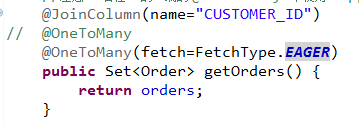
System.***out***.println(customer.getLastName());

System.***out***.println(customer.getOrders().size());

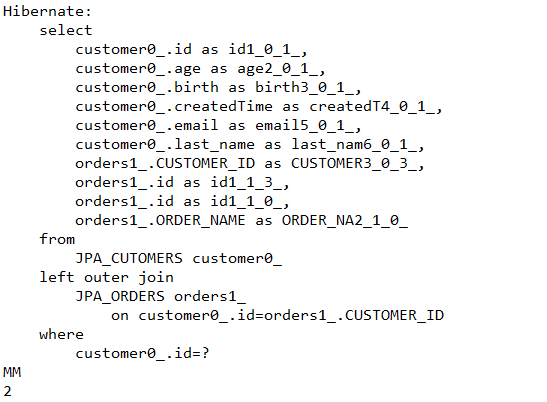
}



### 正常加载（EAGER）



2.可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略



## 删除（一方）

### 外键置空，只删一方

//默认情况下, 若删除 1 的一端, 则会先把关联的 n 的一端的外键置空, 然后进行删除.

//可以通过 @OneToMany 的 cascade 属性来修改默认的删除策略.

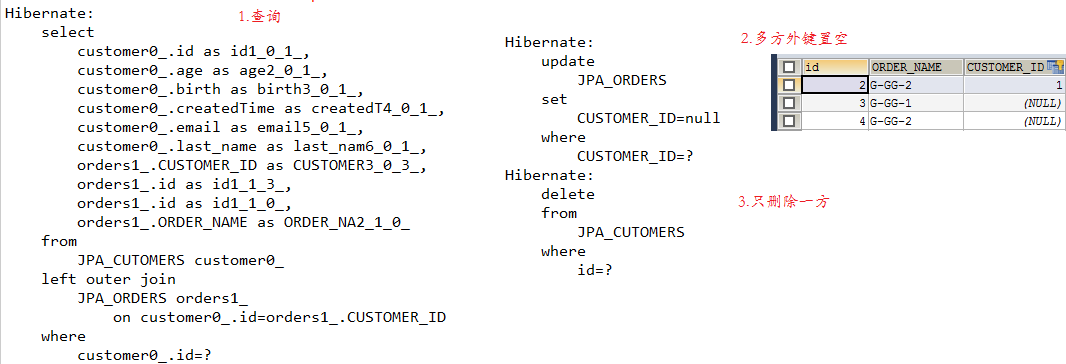
@Test

**public** **void** testOneToManyRemove(){

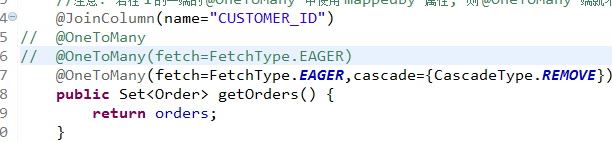
Customer customer = entityManager.find(Customer.**class**, 8);

entityManager.remove(customer);

}

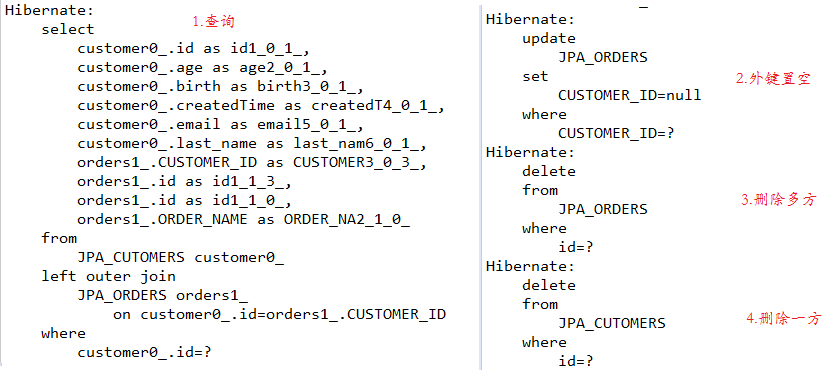


### 外键置空，都删



//可以使用 @OneToMany 的 fetch 属性来修改默认的加载策略

@OneToMany(fetch=FetchType.***EAGER***,cascade={CascadeType.***REMOVE***})



## 更新

只要是针对一方，多方就直接单表了

@Test

**public** **void** testUpdate(){

Customer customer = entityManager.find(Customer.**class**, 3);

customer.getOrders().iterator().next().setOrderName("O-XXX-10");

}

