第 17 章　检索数据 API（上）

在开发应用程序时，常常需要到数据库中检索对象。本章以及第 18 章将系统地介绍 JPA 和 Hibernate 提供的各种检索对象的方式。概括起来，JPA 和 Hibernate 提供了以下几种检索对象的方式。

（1）导航对象图检索方式

根据已经加载的对象，导航到其他对象。例如，对于已经加载的 Customer 对象，调用它的 getOrders().iterator()方法，就可以导航到所有关联的 Order 对象，假如在关联级别使用了延迟加载检索策略，那么首次执行此方法时，Hibernate 会从数据库中加载关联的 Order 对象，否则就从 Session 的持久化缓存中取得 Order 对象。

（2）OID 检索方式

按照对象的 OID 来检索对象。EntityManager 的 getReference()和 find()方法提供了这种功能。如果在应用程序中事先知道了 OID，就可以使用这种检索对象的方式。

（3）JPQL（JPA Query Language）检索方式

使用面向对象的 JPQL 查询语言。这种查询语言和 SQL 语言在语法形式上有些相似，但 JPQL 查询语言中的主体是对象以及其属性。

（4）QBC 检索方式

使用 QBC（Query By Criteria）API 来检索对象。这种 API 封装了基于字符串形式的查询语句，提供了更加面向对象的查询接口。

（5）本地 SQL 检索方式

使用本地数据库的 SQL 查询语句。Hibernate 会负责把检索到的 JDBC ResultSet 结果集映射为持久化对象图。

本章主要介绍 JPA API 提供的 JPQL 检索方式和 QBC 检索方式的基本用法，重点介绍了 JPQL 查询语言的语法，以及 QBC 中主要接口和类的用法。此外还简单介绍了本地 SQL 检索方式的用法。

如果想了解如何通过 Hibernate API 来检索数据，请阅读本章的补充知识，在本书配套光盘上的位置为：book_resource/chapter17。在本书技术支持网站（www.javathinker.net/hibernate.jsp）上也提供了相应的补充知识。

JPQL 是 JPA API 提供的面向对象的查询语句。Hibernate 也提供了 HQL（Hibernate Query Language）查询语句。JPQL 和 HQL 非常相似，本章主要介绍 JPQL。

17.1　检索方式简介

如果直接通过 JDBC API 查询数据库，必须在应用程序中嵌入冗长的 SQL 语句，例如以下代码按照参数指定的客户姓名到数据库中检索匹配的 Customer 对象及关联的 Order 对象：

```
public List<Customer> findCustomerByName(String name) throws Exception{
  HashMap<Long,Customer> map=new HashMap<Long,Customer>();
  List<Customer> result=new ArrayList<Customer>();
```

```
Connection con=null;
PreparedStatement stmt=null;
ResultSet rs=null;
try{
  con=getConnection(); //获得数据库连接
  String sqlString="select c.ID CUSTOMER_ID,c.NAME,c.AGE, "
  +"o.ID ORDER_ID, "
  +"o.ORDER_NUMBER,o.PRICE "
  +"from CUSTOMERS c left outer join ORDERS o"
  +"on c.ID =o.CUSTOMER_ID where c.NAME=?";

  stmt = con.prepareStatement(sqlString);
  stmt.setString(1,name); //绑定参数
  rs=stmt.executeQuery();

  while (rs.next()){
    //编历 JDBC ResultSet 结果集
    Long customerId =Long.valueOf( rs.getLong(1));
    String customerName= rs.getString(2);
    int customerAge= rs.getInt(3);
    Long orderId =Long.valueOf( rs.getLong(4));
    String orderNumber= rs.getString(5);
    double price=rs.getDouble(6);

    //映射 Customer 对象
    Customer customer=null;
    if(map.containsKey(customerId))
      //如果在 map 中已经存在 OID 匹配的 Customer 对象，就获得此对象的引用，
      //这样就避免创建重复的 Customer 对象
      customer=map.get(customerId);
    else{
      //如果在 map 中不存在 OID 匹配的 Customer 对象，
      //就创建一个 Customer 对象，然后把它保存到 map 中
      customer=new Customer();
      customer.setId(customerId);
      customer.setName(customerName);
      customer.setAge(customerAge);
      map.put(customerId,customer);
    }

    //映射 Order 对象
    Order order=new Order();
    order.setId(orderId);
    order.setOrderNumber(orderNumber);
    order.setPrice(price);

    //建立 Customer 对象与 Order 对象的关联关系
    customer.getOrders().add(order);
    order.setCustomer(customer);
  }
  //把 map 中所有的 Customer 对象加入到 result 集合中
  Iterator<Customer> iter =map.values().iterator();
  while ( iter.hasNext() ) {
```

```
            result.add(iter.next());
          }
          return result;
        }finally{
          //关闭 ResultSet 和 Statement 对象
          rs.close();
          stmt.close();
        }
      }
```

假如以上 SQL 语句查询出三条匹配的记录，参见图 17-1。

CUSTOMER_ID	NAME	ORDER_ID	ORDER_NUMBER
1	Tom	1	Tom_Order001
1	Tom	2	Tom_Order002
1	Tom	3	Tom_Order003

图 17-1　SQL 语句查询出的三条记录

应用程序代码负责把 ResultSet 对象中包含的三条记录映射为 Customer 对象和 Order 对象，然后建立它们的关联关系。这三条记录的 CUSTOMER_ID 相同，应该对应同一个 Customer 对象，应用程序采用 HashMap 来避免创建重复的 Customer 对象。

从这个例子可以看出，通过 JDBC API 来查询数据库很麻烦，应用程序必须承担以下职责：

- 定义冗长的基于字符串形式的 SQL 查询语句。
- 把 JDBC ResultSet 中存放的关系数据映射为 Customer 对象和 Order 对象。
- 建立 Customer 和 Order 对象之间的关联关系。
- 确保每个 Customer 对象都具有唯一的 OID。

以下代码通过 JPA 提供的 JPQL 检索方式，按照姓名检索匹配的 Customer 对象及关联的 Order 对象：

```
public List<Customer> findCustomerByName(String name){
  //假定 getEntityManager()方法返回 EntityManager 对象
  EntityManager entityManager=getEntityManager();
  return entityManager.createQuery("from Customer as c "
            +" left join fetch c.orders where c.name='"+name+"'"
              ,Customer.class)
            .getResultList();
}
```

或者：

```
public List<Customer> findCustomerByName(String name) {
  EntityManager entityManager=getEntityManager();

  TypedQuery<Customer> query= entityManager.createQuery(
                  "from Customer as c left join fetch c.orders "
                 +" where c.name=:customerName",Customer.class);
  query.setParameter("customerName",name);
  return query.getResultList();
```

```
}
```

从以上例子可以看出，当应用程序采用 JPQL 检索方式，只需向 JPA API 提供面向对象的 JPQL 查询语句，底层 Hibernate 根据持久化类的对象-关系映射信息，负责把 JPQL 查询语句转换为 SQL 查询语句，并且负责把 JDBC ResultSet 结果集映射为关联的对象图。由此可见，Hibernate 封装了通过 JDBC API 查询数据库的细节。

除了 JPQL 检索方式，JPA 还提供了 QBC 检索方式和本地 SQL 检索方式，下面介绍这些检索方式的特点以及使用场合。

17.1.1 JPQL 检索方式

JPQL（JPA Query Language）是面向对象的查询语言，它和 SQL 查询语言有些相似。在 JPA 提供的各种检索方式中，JPQL 是使用最广的一种检索方式。它具有以下功能：

- 在查询语句中设定各种查询条件。
- 支持投影查询，即仅检索出对象的部分属性。
- 支持分页查询。
- 支持连接查询。
- 支持分组查询，允许使用 having 和 group by 关键字。
- 支持调用各种函数。
- 支持子查询，即嵌入式查询。
- 支持动态绑定参数。

在本书中，“检索”与“查询”其实是一回事。出于表达的便利，“检索”在面向对象的语义中使用得广泛些，而“查询”在面向关系的语义中使用得广泛些。

以下程序代码用于检索姓名为“Tom”，并且年龄为 21 的 Customer 对象：

```
//创建一个 Query 对象
Query query=entityManager.createQuery(
     "from Customer as c where c.name=:customerName "
     +"and c.age=:customerAge");
//动态绑定参数
query.setParameter("customerName","Tom");
query.setParameter("customerAge",21);

//执行查询语句，返回查询结果
List result=query.getResultList();

//遍历访问查询结果
Iterator it=result.iterator();
while(it.hasNext()){
  Customer c=(Customer)it.next();
}
```

从以上程序代码看出，JPQL 检索方式包括以下步骤。

（1）通过 EntityManager 的 createQuery()方法创建一个 Query 对象，它包含一个 JPQL 查询语句。JPQL 查询语句可以包含命名参数，如“customerName”和“customerAge”都是命名参数。

（2）动态绑定参数。Query 接口的 setParameter()方法可以为各种数据类型的命名参数赋值。

（3）调用 Query 的 getResultList()方法执行查询语句。该方法返回 java.util.List 类型的查询结果，在 List 集合中存放了符合查询条件的持久化对象。对于以上程序代码，当运行 Query 的 getResultList()方法时，Hibernate 执行以下 SQL 查询语句：

```
select ID,NAME,AGE from CUSTOMERS where NAME='Tom' and AGE=21;
```

Query 接口支持方法链编程风格，它的 setParameter()方法以及 getResultList()等方法都返回自身实例，而不是返回 void 类型。如果采用方法链编程风格，将按以下形式访问 Query 接口：

```
List result=entityManager.createQuery("……")
                .setParameter("customerName","Tom")
                .setParameter("customerAge",21)
                .getResultList();
```

可见，方法链编程风格能使程序代码更加简洁。

JPQL 与 SQL 在语法形式上有些相似，例如：

```
JPQL 查询语句：from Customer as c where c.name='Tom' and c.age=21
SQL 查询语句：select * from CUSTOMERS where NAME='Tom' and AGE=21;
```

但 JPQL 与 SQL 在本质上是不一样的：

- JPQL 查询语句是面向对象的，Hibernate 负责解析 JPQL 查询语句，然后根据对象-关系映射信息，把 JPQL 查询语句翻译成相应的 SQL 语句。JPQL 查询语句中的主体是域模型中的类以及类的属性。例如在以上 JPQL 查询语句例子中，“Customer”是持久化类的名字，“c.age”是持久化类的属性的名字。
- SQL 查询语句是与关系数据库绑定在一起的。SQL 查询语句中的主体是数据库表以及表的字段。例如在以上 SQL 查询语句例子中，“CUSTOMERS”是表的名字，“AGE”是表的字段的名字。

如果希望明确指定返回结果中对象的类型，可以创建 TypedQuery，TypedQuery 接口继承了 Query 接口：

```
//创建一个 TypedQuery 对象
TypedQuery<Customer> query=entityManager.createQuery(
     "from Customer as c",Customer.class); //设定检索对象的类型

//执行查询语句，返回查询结果
List<Customer> result=query.getResultList();

//遍历访问查询结果
Iterator<Customer> it=result.iterator();
while(it.hasNext()){
  Customer c=it.next(); //无需进行强制类型转换
}
```

本书大部分范例都采用 TypedQuery，使得返回结果可以通过泛型标识<Customer>来明确指定集合中对象的类型。这样做可以避免程序在遍历访问结果集时对每个对象进行强制类型的转换。

17.1.2 QBC 检索方式

采用 JPQL 检索方式时，在应用程序中需要定义基于字符串形式的 JPQL 查询语句。QBC API 提供了检索对象的另一种方式，它提供了一种更加面向对象的查询方法，应用程序不需

要提供查询语句，而是通过 QBC API 中的相关的接口和类来设定需要检索的数据，包括设定检索条件等。

QBC API 位于 javax.persistence.criteria 包中，主要包括以下接口：

- CriteriaBuilder 接口：它是生成 CriteriaQuery 实例的工厂类。
- CriteriaQuery 接口：它是主要的查询接口，通过它来设定需要查询的数据。
- Root 接口：指定需要检索的对象图的根节点对象。
- Selection 接口：指定查询语句。它有个 Expression 子接口，指定查询表达式。
- Expression 接口：指定查询表达式。它有个 Predicate 子接口，指定查询条件。
- Predicate 接口：指定查询条件。

以下程序代码用于检索年龄大于 21 的 Customer 对象：

```
//创建负责生成 CriteriaQuery 的工厂
CriteriaBuilder  criteriaBuilder = entityManager.getCriteriaBuilder();

//创建 CriteriaQuery 对象
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery(Customer.class);

//指定需要检索的对象图的根节点对象
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

//指定查询条件,其中“Customer_”为元数据类
Predicate predicate = criteriaBuilder.gt(
               root.get(Customer_.age), 21);
criteriaQuery.where(predicate);

//到数据库中查询数据，返回查询结果
List<Customer> result=entityManager
       .createQuery(criteriaQuery)
       .getResultList();
```

以上程序代码中的 root.get(Customer_.age)方法访问了元模型类 Customer_，本章 17.1.16 节介绍元模型类的作用和创建方法。root.get(Customer_.age)方法也可以简写为 root.get("age")。

对于以上程序代码，当运行 Query 的 getResultList()方法时，Hibernate 执行的 SQL 查询语句为：

```
select ID,NAME,AGE from CUSTOMERS where AGE>21;
```

从以上程序代码看出，QBC 检索方式包括以下步骤。

（1）创建 CriteriaBuilder 和 CriteriaQuery 对象。

（2）通过 Root 对象指定需要检索的对象图的根节点对象，Hibernate 根据它来决定查询语句中的主表。

（3）通过 Predicate 对象指定查询条件,Hibernate 根据它来决定查询语句中 where 子句的内容。

（4）通过 Query 接口查询数据。

CriteriaBuilder 接口提供了一系列设定查询条件的方法，这些方法都返回 Predicate 对象：

- 表示“等于”的 equal()方法
- 表示“不等于”的 notEqual()方法

- 表示“大于”的 gt()方法
- 表示“大于等于”的 ge()方法
- 表示“小于”的 lt()方法
- 表示“小于等于”的 le()方法
- 表示“位于…之间”的 between()方法
- 表示“相似”的 like()方法
- 表示“不为空”的 isNotEmpty()方法
- 表示“与操作”的 and()方法
- 表示“或操作”的 or()方法

QBC 允许指定多个查询条件。以下程序代码用于检索姓名以字符“T”开头，并且年龄为 21 的 Customer 对象：

```
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

//指定查询条件,其中“Customer_”为元数据类
List<Predicate> predicatesList = new ArrayList<Predicate>();
Predicate predicate1 = criteriaBuilder.like(
                            root.get(Customer_.name),"T%");
Predicate predicate2 = criteriaBuilder.gt(
                            root.get(Customer_.age), 21);
predicatesList.add(predicate1);
predicatesList.add(predicate2);

criteriaQuery.where(predicatesList.toArray(
                new Predicate[predicatesList.size()]));

//到数据库中查询数据，返回查询结果
List<Customer> result=entityManager
       .createQuery(criteriaQuery)
       .getResultList();
```

对于以上程序代码，当运行 Query 的 getResultList()方法时，Hibernate 执行的 SQL 查询语句为：

```
select ID,NAME,AGE from CUSTOMERS where NAME like 'T%' and AGE=21;
```

CriteriaQuery 也支持方法链编程风格，例如：

```
criteriaQuery.select(root)
             .where(predicate);
```

17.1.3 本地 SQL 检索方式

采用 JPQL 或 QBC 检索方式时，Hibernate 会生成标准的 SQL 查询语句，适用于所有的数据库平台，因此这两种检索方式都是跨平台的。

有的应用程序可能需要根据底层数据库的 SQL 方言，来生成一些特殊的查询语句。在这种情况下，可以利用 JPA 提供的本地 SQL 检索方式。以下程序代码用于检索姓名以字符“T”开头，并且年龄为 21 的 Customer 对象：

```
//创建 Query 对象
Query query=entityManager.createNativeQuery (
       "select * from CUSTOMERS where NAME like :customerName "
```

```
            +"and AGE=:customerAge");

        //动态绑定参数
        query.setParameter("customerName","T%");
        query.setParameter("customerAge",21);

        //执行 SQL select 语句，返回查询结果
        List result=query.getResultList();
```

从以上程序代码看出，本地 SQL 检索方式与 JPQL 检索方式都使用 Query 接口，区别在于本地 SQL 检索方式通过 EntityManager 的 createNativeQuery()方法来创建 Query 对象，这个方法的参数指定一个 SQL 查询语句，该语句可以使用本地数据库的 SQL 方言。第 18 章的 18.4.4 节（本地 SQL 查询）还会进一步介绍本地 SQL 检索方式的用法。

17.1.4 关于本章范例程序

本章范例程序位于配套光盘的 sourcecode/chapter17 目录下。该范例程序并没有包含本章涉及的所有演示代码，仅仅为读者提供了一个便于测试本章演示代码的运行环境。读者需要按照 schema 子目录下的 SQL 脚本文件 sampledb.sql，在数据库中手工创建 CUSTOMERS 表和 ORDERS 表，再加入测试数据。随后把本章的演示代码加入到 BusinsessService 类的 findAny()方法中，就可以通过 ant run 命令运行这个类。

CUSTOMERS 表包含如图 17-2 所示的数据。ORDERS 表包含如图 17-3 所示的数据。

ID	NAME	AGE
1	Tom	21
2	Mike	24
3	Jack	30
4	Linda	25
5	Tom	25

图 17-2　CUSTOMERS 表包含的数据

ID	ORDER_NUMBER	PRICE	CUSTOMER_ID
1	Tom_Order001	100.00	1
2	Tom_Order002	200.00	1
3	Tom_Order003	300.00	1
4	Mike_Order001	100.00	2
5	Jack_Order001	200.00	3
6	Linda Order001	100.00	4
7	UnknownOrder	200.00	NULL

图 17-3　ORDERS 表包含的数据

如果没有特别说明，本章列举的查询语句都是针对以上表进行查询的，所得到的查询结果都建立在这些表的数据的基础上。

17.1.5 使用别名

以下代码通过最简单的 JPQL 查询语句检索 Customer 持久化类的所有实例：

```
//采用 JPQL 检索方式
List<Customer> result=EntityManager
    .createQuery("from Customer",Customer.class)
    .getResultList();
```

通过 JPQL 检索一个类的实例时，如果查询语句的其他地方需要引用它，应该为这个类指定一个别名，例如：

```
from Customer as c where c.name=:name
```

as 关键字用于设定别名，也可以将 as 关键字省略：

```
from Customer c where c.name=:name
```

在实际应用中，建议使别名与类名相同，例如为 Customer 类赋予别名“customer”，而不是别名“c”：

```
from Customer as customer where customer.name=:name
```

本书为了保持版面的简洁，把“customer”简写为“c”。此外，本书对 JPQL 查询语句中的关键字一律采用小写形式，事实上，JPQL 查询语句中的关键字（如“from”、“as”和“where”等）不区分大小些， 例如以下 JPQL 查询语句也是合法的：

```
FROM Customer AS customer WHERE customer.name=:name
```

QBC 检索方式一般不需要由应用程序显式指定类的别名，QBC 检索方式会通过元模型类来引用查询语句中的类，例如：

```
Predicate predicate = criteriaBuilder.gt(root.get(Customer_.age), 21);
criteriaQuery.where(predicate);
```

对于 QBC 检索方式，在某些场合，如果需要设定类的别名，可以采用以下方式。本书第 18 章的 18.1.11 节会介绍使用别名的范例：

```
Root<Customer> root = criteriaQuery.from(Customer.class );
Selection<Customer> rootAlias=root.alias("c");  //设置别名为 c
```

17.1.6 多态查询

JPQL 和 QBC 都支持多态查询，多态查询是指查询出当前类以及所有子类的实例，对于以下查询代码:

```
//采用 JPQL 检索方式
entityManager.createQuery("from Employee");

//采用 QBC 检索方式
CriteriaQuery<Employee> criteriaQuery =
     criteriaBuilder.createQuery(Employee.class);
Root<Employee> root = criteriaQuery.from(Employee.class);
criteriaQuery.select(root);
```

假如 Employee 有两个子类：HourlyEmployee 和 SalariedEmployee，那么这个查询语句会查询出所有 Employee 类的实例，以及 HourlyEmployee 类和 SalariedEmployee 类的实例。如果只想检索某个特定子类的实例，可以使用如下方式：

```
//采用 JPQL 检索方式
entityManager.createQuery("from HourlyEmployee");

//采用 QBC 检索方式
CriteriaQuery<HourlyEmployee> criteriaQuery =
     criteriaBuilder.createQuery(HourlyEmployee.class);
Root<HourlyEmployee> root = criteriaQuery.from(HourlyEmployee.class);
criteriaQuery.select(root);
```

以下 JPQL 查询语句将检索出所有的持久化对象：

```
from java.lang.Object
```

多态查询对接口也适用。例如，以下 JPQL 查询语句检索出所有实现 Serializable 接口的实例:

```
from java.io.Serializable
```

JPQL 查询语句不仅对 from 子句中显式指定的类进行多态查询，而且对这个类的所关联的类也会进行多态查询。

对于 QBC，以下代码试图检索出所有的持久化对象：

```
CriteriaQuery<Object> criteriaQuery =
        criteriaBuilder.createQuery( Object.class);
Root<Object> root = criteriaQuery.from(Object.class);
criteriaQuery.select(root);
```

运行以上代码会抛出 IllegalArgumentException 异常：

```
Exception in thread "main" java.lang.IllegalArgumentException:
Not an entity: class java.lang.Object
```

这是因为 QBC 能够检索的类必须使用了@Entity 注解，表明是实体类。

17.1.7 对查询结果排序

JPQL 与 QBC 都支持对查询结果排序。JPQL 采用“order by”关键字对查询结果排序，而 QBC 采用 javax.persistence.criteria.Order 类来指定对查询结果的排序方式，CriteriaBuilder 接口的 asc()方法和 desc()方法分别返回表示升序或降序的 Order 对象。下面举例说明它们的用法。

（1）查询结果按照客户姓名升序排列：

```
//JPQL 检索方式
Query query=entityManager
       .createQuery("from Customer c order by c.name");

//QBC 检索方式
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

javax.persistence.criteria.Order order=
  criteriaBuilder.asc(root.get(Customer_.name));
criteriaQuery.orderBy(order);  //指定排序方式
```

由于本章范例中也有一个 mypack.Order 类，为了避免和 javax.persistence.criteria.Order

类混淆，所以在以上程序代码中引用 javax.persistence.criteria.Order 类的全名。

如果程序必须调用 CriteriaQuery 的 select()、where()和 orderBy()方法，那么必须按照这三个方法在这里列出的先后顺序来调用。

（2）查询结果按照客户姓名升序排列，并且按照年龄降序排列：

```
//JPQL 检索方式
Query query=entityManager.createQuery(
       "from Customer c order by c.name asc,c.age desc");

//QBC 检索方式
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery(Customer.class);
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

javax.persistence.criteria.Order order1=
            criteriaBuilder.asc(root.get(Customer_.name));
javax.persistence.criteria.Order order2=
            criteriaBuilder.desc(root.get(Customer_.age));

List<javax.persistence.criteria.Order> ordersList=new LinkedList<>();
ordersList.add(order1);
ordersList.add(order2);
criteriaQuery.orderBy(ordersList);
```

17.1.8 分页查询

当批量查询数据时（例如查询 CUSTOMERS 表中所有记录），如果数据量很大，会导致无法在用户终端的单个页面上显示所有的查询结果，此时需要对查询结果分页。假如 CUSTOMER 表中有 99 条记录，可以在用户终端上分 10 页来显示结果，每一页最多只显示 10 个 Customer 对象，用户既可以导航到下一页，也可以导航到前一页。Query 接口提供了用于分页获取查询结果的方法：

- setFirstResult(int firstResult)：设定从哪一个对象开始检索，参数 firstResult 表示这个对象在查询结果中的索引位置，索引位置的起始值为 0。在默认情况下，Query 接口从查询结果中的第一个对象，也就是索引位置为 0 的对象开始检索。
- setMaxResult(int maxResults)：设定一次最多检索出的对象数目。在默认情况下，Query 接口检索出查询结果中所有的对象。

以下代码从查询结果的起始对象开始，共检索出 10 个 Customer 对象，查询结果按照 name 属性排序：

```
//采用 JPQL 检索方式
TypedQuery<Customer> query = entityManager.createQuery(
           "from Customer c order by c.name asc",Customer.class);
query.setFirstResult(0);
query.setMaxResults(10);
List<Customer> result = query.getResultList();

//采用 QBC 检索方式
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery(Customer.class);
```

```
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

javax.persistence.criteria.Order order=
  criteriaBuilder.asc(root.get(Customer_.name));
criteriaQuery.orderBy(order);  //指定排序方式

TypedQuery<Customer> query=entityManager.createQuery(criteriaQuery);
query.setFirstResult(0);
query.setMaxResults(10);
List<Customer> result=query.getResultList();
```

如果查询结果中共有 99 个 Customer 对象，那么在 result 中包含 10 个 Customer 对象，第 1 个对象在查询结果中的索引位置为 0，第 10 个对象在查询结果中的索引位置为 9。

以下代码从查询结果中索引位置为 97 的对象开始，共检索出 10 个 Customer 对象：

```
//采用 JPQL 检索方式
TypedQuery<Customer> query = entityManager.createQuery(
             "from Customer c order by c.name asc",Customer.class);
query.setFirstResult(97);
query.setMaxResults(10);
List<Customer> result = query.getResultList();

//采用 QBC 检索方式
……
TypedQuery<Customer> query=entityManager.createQuery(criteriaQuery);
query.setFirstResult(97);
query.setMaxResults(10);
List<Customer> result = query.getResultList();
```

如果查询结果中共有 99 个对象，那么在 result 中只包含两个 Customer 对象，它们的索引位置分别为 97 和 98。

17.1.9 检索单个对象（getSingleResult()方法）

Query 接口提供了以下用于执行查询语句并返回查询结果的方法。

- getResultList()方法：返回一个 List 类型的查询结果，在 List 集合中存放了所有满足查询条件的对象。
- getSingleResult()方法：返回单个对象。

在某些情况下，如果只希望检索出一个对象，可以先调用 Query 接口的 setMaxResults(1)方法，把最大检索数目设为 1：

```
setMaxResults(1);
```

接下来调用 getSingleResult()方法，该方法返回一个 Object 类型的对象：

```
//采用 JPQL 检索方式
TypedQuery<Customer> query = entityManager.createQuery(
             "from Customer c order by c.name asc",Customer.class);
query.setMaxResults(1);
Customer customer =query.getSingleResult();

//采用 QBC 检索方式
……
```

```
TypedQuery<Customer> query=entityManager.createQuery(criteriaQuery);
query.setMaxResults(1);
Customer customer =query.getSingleResult();
```

如果明确知道查询结果只会包含一个对象，可以不调用 setMaxResults(1)方法，例如：

```
Customer customer =entityManager
                   .createQuery("from Customer c where c.id=1",
                     Customer.class)
                   .getSingleResult();
```

以下查询结果会包含多个 Customer 对象，但是没有调用 setMaxResults(1)方法:

```
Customer customer =entityManager
             .createQuery("from Customer c order by c.name asc",
                Customer.class)
             .getSingleResult();
```

执行以上 getSingleResult()方法时，会抛出 NonUniqueResultException 异常：

```
javax.persistence.NonUniqueResultException:
query did not return a unique result
```

如果查询结果中不包含任何 Customer 对象，那么 Query 的 getSingleResult()方法会抛出 NoResultException 异常。

17.1.10 按主键挨个处理查询结果（属于 Hibernate 的功能）

JPA API 和 Hibernate API 都有 Query 接口：分别是 javax.persistence.Query 接口和 org.hibernate.query.Query 接口。javax.persistence.Query 接口的 getResultList()方法与 org.hibernate.query.Query 接口的 getResultList()方法的作用相同，都返回包含了所有查询结果的 List 对象。

org.hibernate.query.Query 接口继承了 javax.persistence.Query 接口。因此，org.hibernate.query.Query 接口也有 getResultList()方法。此外，org.hibernate.query.Query 接口还有一个 list()方法，该方法的作用与 getResultList()方法相同。

在 Hibernate 的早期版本中，Hibernate API 的 org.hibernate.query.Query 接口还提供了一个 iterate()方法，在 Hibernate5 以上版本中，该方法已经被淘汰。本章对该方法的作用还是做一个简单介绍。iterate()方法和 getResultList()方法一样，也能执行查询操作。iterate()和 getResultList()方法的区别在于两者使用的查询机制不一样。

对于以下程序代码：

```
Session session=entityManager.unwrap(Session.class);  //第 1 行

List<Customer> customers1=session
    .createQuery("from Customer c  where  c.age>10",Customer.class)
    .getResultList();  //第 2 行

Iterator<Customer> customers2=session
     .createQuery("from Customer c  where c.age>10",Customer.class)
     .iterate();  //第 3 行
```

第 2 行程序的 getResultList()方法执行的 SQL select 语句中包含 CUSTOMERS 表中所有字段：

```
select ID,NAME,AGE from CUSTOMERS where AGE>10;
```

第 3 行程序的 iterate()方法执行的 SQL 语句中仅仅包含 CUSTOMERS 表中 ID 字段：

```
select ID from CUSTOMERS where AGE>10;
```

对于以下程序代码：

```
//先把 ID 为 1 的 Customer 对象加载到 Session 的持久化缓存中
Customer customer1=session.get(Customer.class,
                                    Long.valueOf(1));
Iterator<Customer> customers =session
     .createQuery("from Customer c  where c.age>10",Customer.class)
     .iterate();

while(customers.hasNext()){
  Customer customer=customers.next();
  System.out.println(customer.getName());
}
```

当每次通过 customers.next()方法来遍历访问结果集时，该方法先到 Session 的持久化缓存以及 Hibernate 的第二级缓存（在已经启用了第二级缓存的情况下）中查看是否存在拥有特定 OID 的 Customer 对象，如果存在，就直接返回该对象，否则就通过相应的 SQL select 语句到数据库中加载特定的 Customer 对象。

对于以上程序代码，当第一次执行 customers.next()方法时，访问的是 OID 为 1 的 Customer 对象，由于 Session 的持久化缓存中已经存在该对象，因此不必访问数据库。

当第二次执行 customers.next()方法时，访问的是 OID 为 2 的 Customer 对象，由于 Session 的持久化缓存中不存在该对象，因此通过以下 SQL select 语句到数据库中加载 Customer 对象：

```
select ID,NAME,AGE from CUSTOMERS where ID=2;
```

假定对于 JPQL 查询语句“from Customer c where c.age>10”，共有 n 个符合查询条件的 Customer 对象，并且假定执行 iterate()方法之前，Session 的持久化缓存以及第二级缓存中不存在任何符合查询条件的 Customer 对象，那么通过 iterate()方法以及接下来的 customers.next()方法遍历访问结果集时，一共会执行 n+1 条 SQL select 语句。

在大多数情况下，应该考虑用 org.hibernate.query.Query 的 getResultList()方法（或者 javax.persistence.Query 的 getResultList()方法）来执行查询操作。org.hibernate.query.Query 的 iterate()方法仅仅在满足以下条件的场合，可以稍微提高查询的性能：

（1）CUSTOMERS 表中包含大量的字段。

（2）启用了 Hibernate 的第二级缓存，第二级缓存中可能已经包含待查询的 Customer 对象。

在满足上述两个条件的情况下，iterate()方法以及接下来的 customers.next()方法有可能只要通过一个 SQL select 语句就可以检索出所有满足查询条件的 Customer 对象：

```
select ID from CUSTOMERS where AGE>10;
```

由于以上 select 语句不必包含 CUSTOMERS 表中所有字段，而仅仅包含 ID 字段，因此可以稍微提高查询性能。

另外值得注意的是，org.hibernate.query.Query 的 iterate()方法返回的 Iterator 对象一直处于打开状态，在以下情况被关闭：

（1）遍历访问完结果集中的所有对象。

（2）关闭 Session 对象。

（3）通过 org.hibernate.Hibernate.close(iterator)方法关闭 Iterator 对象。

17.1.11 可滚动的结果集（属于 Hibernate 的功能）

JDBC API 提供了一种可滚动的结果集，它是利用数据库系统中的游标来实现的。游标用于定位查询结果集中的记录，应用程序可以通过任意移动游标来定位到特定记录。

在 Hibernate 的早期版本中，Hibernate API 中的 org.hibernate.Query 接口的 scroll()方法返回一个 org.hibernate.ScrollableResults 对象，它就代表可滚动的结果集。在 Hibernate5 以上版本中，该方法已经被淘汰，本章还是对该方法的作用做一个简单的介绍。org.hibernate.ScrollableResults 接口与 java.sql.ResultSet 接口有一些相似，但不完全相同。ScrollableResults 接口包含以下用于移动游标的方法：

- first()：使游标移动到第一行。
- last()：使游标移动到最后一行。
- beforeFirst()：使游标移动到结果集的开头（第一行之前）。
- afterLast()：使游标移动到结果集的末尾（最后一行之后）。
- previous()：使游标从当前位置向上（或者说向前）移动一行。
- next()：使游标从当前位置向下（或者说向后）移动一行。
- scroll(int n)：使游标从当前位置移动 n 行。如果 n>0，就向下移动，否则就向上移动。当 n 为 1，等价于调用 next()方法；当 n 为-1，等价于调用 previous()方法。
- setRowNumber(int n): 使游标移动到行号为 n 的行。参数 n 指定行号。结果集的行号从 0 开始编号。如果 n 为 0，使游标移动到第一行；如果 n 为 2，使游标移动到第三行；如果 n 为-1，使游标移动到最后一行。

在以上方法中，除了 beforeFirst()和 afterLast()方法返回 void 类型，其余方法都返回 boolean 类型，如果游标移动到的目标位置具有记录，就返回 true，否则返回 false。

如图 17-4 所示，假定 SQL 语句“select ID,NAME,AGE from CUSTOMERS”从 CUSTOMERS 表中共查出五条记录。

游标 →

行号	ID	NAME	AGE
0	1	Tom	21
1	2	Mike	24
2	3	Jack	30
3	4	Linda	25
4	5	Tom	25

图 17-4 用游标来定位结果集中的记录

以下程序代码演示了 ScrollableResults 接口的基本用法。

```
Session session=entityManager.unwrap(Session.class);

ScrollableResults rs=session.createQuery("from Customer c ")
                            .scroll();
//游标移动到结果集中的第一行
rs.first();

//获得当前行中的所有字段，在本例中，当前行只有一个字段，为 Customer 对象
//因此返回的对象数组中只有一个 Customer 对象
Object[] o=rs.get();
Customer customer=(Customer)o[0];  //获取对象数组中的第 1 个对象
System.out.println(customer.getId()
```

```
        +" "+customer.getName()+" "+customer.getAge());

    rs.scroll(2);  //游标从当前位置移动 2 行，即移动到第 3 行
    customer=(Customer)rs.get(0); //获得当前行中的第 1 个字段，为 Customer 对象
    System.out.println(customer.getId()
          +" "+customer.getName()+" "+customer.getAge());

    rs.close();  //关闭结果集
```

以下程序代码演示了 ScrollableResults 接口在分页处理数据时的用法。

```
    final int PAGE_SIZE=3; //每一页处理 3 个 Customer 对象
    List firstNamesOfPages =new ArrayList(); //存放所有页中的第一个客户的姓名
    List pageOfCustomers =new ArrayList(); //存放第一页的所有 Customer 对象

    //以下 HQL 语句中采用了 select 关键字，为投影查询
    ScrollableResults rs=session.createQuery(
                          "select c.name,c from Customer c ")
                           .scroll();

    if ( rs.first() ){

      //演示遍历访问每一页的特定内容
      //获取所有页中的第一个客户的姓名
      do{
          String name = rs.getString(0); //获得当前行的第 1 个字段，为 name 字段
          firstNamesOfPages.add(name);
      }while ( rs.scroll(PAGE_SIZE) );

      //演示访问特定页的所有内容
      //获取第一页的所有 Customer 对象
      rs.beforeFirst();
      int i=0;
      while( ( PAGE_SIZE > i++ ) && rs.next() ){
        //获得当前行的第 2 个字段，为 Customer 对象
        pageOfCustomers.add(rs.get(1));
      }
    }
    rs.close();  //关闭结果集

    for(int i=0;i<firstNamesOfPages.size();i++)
      System.out.println(firstNamesOfPages.get(i));

    for(int i=0;i<pageOfCustomers.size();i++)
      System.out.println(((Customer)pageOfCustomers.get(i)).getName());
```

org.hibernate.Query 接口的 scroll()方法还可以包含一个用于设置滚动模式的参数，例如：

```
    ScrollableResults rs=session.createCriteria(Customer.class)
                               .scroll(ScrollMode.FORWARD_ONLY);
```

org.hibernate.ScrollMode 类提供了以下表示结果集滚动模式的静态常量：

- ScrollMode.FORWARD_ONLY：游标只能从上往下移动，即结果集不能滚动。这是默认值。
- ScrollMode.SCROLL_INSENSITIVE：游标可以上下移动，即结果集可以滚动。当

程序对结果集的内容作了修改，游标对此不敏感。

- ScrollMode. SCROLL_SENSITIVE：游标可以上下移动，即结果集可以滚动。当程序对结果集的内容作了修改，游标对此敏感。比如当程序删除了结果集中的一条记录时，游标位置会随之发生变化。

只有在始终保持数据库连接的情况下，才可以访问 ScrollableResults 结果集中的数据。在访问 ScrollableResults 结果集结束后，应该及时调用 ScrollableResults.close()方法关闭结果集，从而关闭数据库中与此结果集对应的游标。

17.1.12 绑定参数

对于实际应用，经常有这样的需求，用户在查询窗口中输入一些查询条件，要求返回满足查询条件的记录。例如用户提供了姓名和年龄信息，要求查询匹配的 Customer 对象。应用程序可以定义一个 findCustomers()方法来提供这一功能：

```
public List<Customer> findCustomers(String name,int age){
  //假定 getEntityManager()方法返回 EntityManager 对象
  EntityManager entityManager=getEntityManager();

  TypedQuery<Customer> query=entityManager.createQuery(
        "from Customer as c where c.name='"+name+"'"
        +" and c.age="+age,Customer.class);
  retun query.getResultList();
}
```

以上程序代码尽管是可行的，但是不安全，假如有个不怀好意的用户在查询窗口的姓名输入框中输入以下内容：

```
Tom' and SomeStoredProcedure() and 'hello'='hello
```

那么实际的 JPQL 查询语句为：

```
from Customer as c
where c.name='Tom' and SomeStoredProcedure() and 'hello'='hello'
and c.age=20
```

以上查询语句不仅会执行数据库查询，而且会执行一个名为“SomeStoredProcedure”的存储过程。怀有恶意的用户可以通过这种方式来非法调用数据库系统的存储过程。

JPA 采用参数绑定机制来避免以上问题，JPA 的参数绑定机制依赖于 JDBC API 中 PreparedStatement 的预定义 SQL 语句功能。总的说来，参数绑定机制有以下优点：

- 非常安全，防止怀有恶意的用户非法调用数据库系统的存储过程。
- 能够利用底层数据库预编译 SQL 语句的功能，提高查询数据的性能。预编译是指底层数据库系统只需编译 SQL 语句一次，把编译出来的可执行代码保存在缓存中，如果多次执行相同形式的 SQL 语句，不需要重新编译，只要从缓存中获得可执行代码即可。

1．参数绑定的形式

JPQL 的参数绑定有两种形式。

（1）按参数名字绑定。

在 JPQL 查询语句中定义命名参数，命名参数以“:”开头，形式如下：

```
TypedQuery<Customer> query=entityManager.createQuery(
     "from Customer as c where c.name=:customerName "
     +"and c.age=:customerAge",Customer.class);
```

以上 JPQL 查询语句定义了两个命名参数“customerName”和“customerAge”。接下来调用 Query 的 setParameter()方法来绑定参数：

```
query.setParameter("customerName",name);
query.setParameter("customerAge",age);
```

Query 的 setParameter()方法能够绑定各种数据类型的命名参数。该方法的第一个参数代表命名参数的名字，第二个参数代表命名参数的值。

假如有个不怀好意的用户在搜索窗口的姓名输入框中输入以下内容：

```
Tom ' and SomeStoredProcedure() and 'hello'='hello
```

JPA 会把以上字符串中的单引号解析为普通的字符，在 JPQL 查询语句中用两个单引号表示：

```
from Customer as c
where c.name='Tom'' and SomeStoredProcedure() and ''hello''=''hello'
and c.age=20
```

由此可见，参数绑定能够有效的避免本节开头提出的安全漏洞。

在 SQL 语句中，如果字符串中包含单引号，应该采用重复单引号的形式，例如：

update CUSTOMERS set NAME="'**Tom**' where ID=1;

以上 update 语句把 NAME 字段的值改为：'Tom

（2）按参数位置绑定。

在 JPQL 查询语句中用“?”来定义参数的位置，形式如下：

```
TypedQuery<Customer> query=entityManager.createQuery(
      "from Customer as c where c.name=?1 "
      +"and c.age=?2 ",Customer.class);
```

以上 JPQL 查询语句定义了两个参数，第一个参数的位置为 1，第二个参数的位置为 2。接下来调用 Query 的 setParameter()方法来绑定参数：

```
query.setParameter(1,name);
query.setParameter(2,age);
```

Query 的 setParameter()方法的第 1 个参数代表 JPQL 查询语句中参数的位置，第 2 个参数代表 JPQL 查询语句中参数的值。

比较按名字绑定和按位置绑定这两种绑定参数的形式，按名字绑定方式有以下优势：

- 使程序代码有较好的可读性。
- 按名字绑定方式有利于程序代码的维护，而对于按位置绑定方式，如果参数在 JPQL 查询语句中的位置改变了，就必须修改相关绑定参数的代码，这削弱了程序代码的健壮性和可维护性。例如以下程序代码交换了 name 和 age 参数的位置，那么 Query 的 setParameter()方法的参数也要进行相应的调整：

```
TypedQuery<Customer> query=entityManager.createQuery(
  "from Customer as c where c.age=?1 "
  +"and c.name=?2 ",Customer.class);
query.setParameter(2,name);
query.setParameter(1,age);
```

- 按名字绑定方式允许一个参数在 JPQL 查询语句中出现多次，例如：

```
from Customer as c
where c.name like :stringMode and c.email like :stringMode
```

由此可见，应该优先考虑使用按名字绑定方式。

2．绑定时间类型的参数

Query 接口的 setParameter()方法能绑定各种类型的参数，不过，对于时间类型的参数，需要指定具体的时间类型。例如，假定 Customer 类有一个表示生日的 birthday 属性。以下代码检索生日大于特定日期的 Customer 对象：

```
Date oneDay=…… ;
List<Customer> result=entityManager
   .createQuery("from Customer c where c.birthday>:someDay",
     Customer.class)
   .setParameter("someDay",oneDay,TemporalType.DATE)
   .getResultList();
```

以上 javax.persistence.TemporalType 类指定具体的时间类型，它有三个常量：TemporalType.DATE（表示日期类型）、TemporalType.TIME（时间类型）和 TemporalType.TIMESTAMP（时间戳类型）。

3．绑定实体类型的参数

Query 接口的 setParameter()方法还能绑定实体类型的参数。例如以下 setParameter()方法把“customer”命名参数与一个 Customer 对象绑定：

```
List<Customer> result=entityManager
      .createQuery("from Order o where o.customer = :customer",
        Customer.class)
      .setParameter("customer", customer)
      .getResultList();
```

假定以上 customer 变量所引用的 Customer 对象的 OID 为 1，那么 Hibernate 执行的 SQL 查询语句为：

```
  select * from ORDERS where CUSTOMER_ID=1 ;
```

4．参数绑定对 null 安全

参数绑定对 null 是安全的，例如以下程序代码不会抛出异常:

```
String name=null;
List<Customer> result=entityManager
      .createQuery("from Customer c where c.name = :name",
         Customer.class)
      .setString("name", name)
      .getResultList();
```

以上 JPQL 查询语句对应的 SQL 查询语句为：

```
select * from CUSTOMERS where NAME=null;
```

这条查询语句的查询结果永远为空。如果要查询名字为 null 的客户，应该使用“is null”比较运算符 ，参见本章 17.2.1 节。

5．通过 QBC 定义命名参数

CriteriaBuilder 接口的 parameter()方法用来定义一个命名参数，该方法返回一个 ParameterExpression 对象，表示命名参数：

```
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery( Customer.class );
Root<Customer> root = criteriaQuery.from(Customer.class );
```

```
criteriaQuery.select(root);

//定义命名参数
ParameterExpression<String> customerNameParameter=
   criteriaBuilder.parameter(String.class,"customerName");

//设置查询条件
Predicate predicate=criteriaBuilder.equal(root.get("name"),
                                    customerNameParameter);
criteriaQuery.where(predicate);

List<Customer> result=entityManager
      .createQuery(criteriaQuery);
      .setParameter("customerName","Mike") //绑定参数
      .getResultList();
```

以上程序代码也可以按照方法链编程风格简写为：

```
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery( Customer.class );
Root<Customer> root = criteriaQuery.from(Customer.class );

List<Customer> result= entityManager.createQuery(
   criteriaQuery
      .select(root)
      .where(
        criteriaBuilder.equal(
          root.get("name"),
          criteriaBuilder.parameter(String.class,"customerName"))
       )
    ).setParameter("customerName","Mike") //绑定参数
      .getResultList();
```

17.1.13 设置查询附属事项

在采用 JPQL 检索方式或者 QBC 检索方式来检索数据时，可以通过 Query 接口的一些方法来设定查询附属事项：

- setFlushMode()方法：设置清理缓存的模式。
- setHint()方法：设置访问 Hibernate 的第二级缓存的模式、查询超时时间、批量抓取数目、查询注解等。

1．设置清理缓存模式

本书第 8 章的 8.2.2 节（脏检查及清理缓存的机制）已经讲过，当 Session 清理持久化缓存时，会先进行脏检查，即比较 Customer 持久化对象的当前属性与它的快照，来判断 Customer 对象的属性是否发生了变化，如果发生了变化，就称这个对象是“脏对象”，Session 会根据脏对象的最新属性来执行相关 SQL 语句，从而同步更新数据库。JPA 用 FlushModeType 类来表示清理模式，它有两个常量：FlushModeType.COMMIT 和 FlushModeType.AUTO。

以下程序代码把清理模式设为 FlushModeType.COMMIT，因此 Hibernate 在执行 SQL 查询语句之前不会先清理缓存：

```
TypedQuery<Customer> query=entityManager
                .createQuery("from Customer c",
```

```
                    Customer.class)
                .setFlushMode(FlushModeType.COMMIT);
```

2．设置 Hibernate 访问第二级缓存的模式

本书第 22 章的 22.4.6 节（设置第二级缓存的读写模式）会介绍 Hibernate 访问第二级缓存的模式，包括：

- 读缓存模式：决定在检索数据时是否使用第二级缓存。用 CacheRetrieveMode 类来表示，有两个常量：CacheRetrieveMode.BYPASS 和 CacheRetrieveMode.USE。
- 写缓存模式：决定在保存数据时是否把数据保存到第二级缓存中。用 CacheStoreMode 类来表示，有三个常量：CacheStoreMode.BYPASS、CacheStoreMode.USE 和 CacheStoreMode.REFRESH。

关于以上各种模式的具体含义，请参阅第 22 章的 22.4.6 节。以下代码把访问第二级缓存的读模式设为 CacheRetrieveMode.USE，把写模式设为 CacheStoreMode.REFRESH：

```
List<Customer> result=entityManager
      .createQuery("from Customer",Customer.class)
      //设置对第二级缓存的读模式
      .setHint("javax.persistence.cache.retrieveMode",
               CacheRetrieveMode.USE)
      //设置对第二级缓存的写模式
      .setHint("javax.persistence.cache.storeMode",
               CacheStoreMode.REFRESH)
     .getResultList();
```

3．设置执行查询的超时时间

以下程序代码通过 Query 的 setHint()方法，把执行查询数据库操作的超时时间设为 1 分钟：

```
List<Customer> result=entityManager
   .createQuery("from Customer",Customer.class)
     //设置执行查询数据库操作的超时时间为 60 秒
   .setHint("javax.persistence.query.timeout",
            60000)
   .getResultList();
```

4．设置批量抓取数目

以下程序代码通过 Query 的 setHint()方法，把底层 JDBC 驱动程序进行批量抓取的数目设为 50：

```
List<Customer> result=entityManager
      .createQuery("from Customer",Customer.class)
      //设置批量抓取数目
      .setHint(org.hibernate.annotations.QueryHints.FETCH_SIZE,50)
      .getResultList();
```

以上 Query 接口的 setHint()方法和 JDBC API 中 java.sql.Statement 接口的 setFetchSize()方法的作用是一样的，都是为底层 JDBC 驱动程序设置批量抓取数目。

接下来解释一下为底层 JDBC 驱动程序设置批量抓取数目的意义。以下程序代码通过 JDBC API 查询 CUSTOMERS 表中的所有记录：

```
ResultSet rs= stmt.executeQuery(
         "SELECT ID,NAME,AGE,ADDRESS from CUSTOMERS");
while(rs.next()){
```

```
  String id=rs.getLong(1);
  ……
}
```

假如 CUSTOMERS 表中有 100000 条记录，那么 java.sql.Statement 对象的 executeQuery()方法返回的 java.sql.ResultSet 对象中是否会立即存放了这 100000 条记录呢？假如 ResultSet 对象中存放了这么多记录，那将消耗多大内存空间啊。幸运的是，ResultSet 对象实际上并不会包含这么多数据，只有当程序遍历结果集时，ResultSet 对象才会到数据库中抓取相应的数据。ResultSet 对象抓取数据的过程对程序完全是透明的。

那么，是否每当程序访问结果集中的一条记录时，ResultSet 对象就到数据库中抓取一条记录呢？按照这种方式抓取大量记录需要频繁地访问数据库，显然效率很低。为了提高减少访问数据库的次数，JDBC 希望 ResultSet 接口的实现能支持批量抓取，即每次从数据库中抓取多条记录，把它们存放在 ResultSet 对象的缓存中，让程序慢慢享用。在 java.sql.Connection、java.sql.Statement 和 java.sql.ResultSet 接口中都提供了 setFetchSize(int size)方法，用于设置批量抓取的数目。

5．设置查询注释

在对程序代码进行优化时，往往需要查阅和分析复杂的 SQL 日志，为了提高 SQL 日志的可读性，可以在 SQL 日志中加入适当的注释，步骤如下。

（1）把 Hibernate 的配置属性 hibernate.use_sql_comments 设为 true：

```
//在 Hibernate 的 Java 属性格式的配置文件 hibernate.properties 中设置
hibernate.use_sql_comments=true

//在 Hibernate 的 XML 格式的配置文件 hibernate.cfg.xml 中设置
<property name="use_sql_comments">true</property>

//在 JPA 的 XML 格式的配置文件 persistence.xml 中设置
<property name="hibernate.use_sql_comments" value="true" />
```

（2）在程序代码中通过 Query 的 setHint()方法为 SQL 日志加入适当的注释：

```
List<Customer> result=entityManager
      .createQuery("from Customer",Customer.class)
      //设置注解
      .setHint(org.hibernate.annotations.QueryHints.COMMENT,
               "My Comment...")
      .getResultList();
```

运行以上程序代码时，将打印如下带注释的 SQL 日志：

```
Hibernate: /* My Comment... */
select this_.ID as ID0_0_, this_.NAME as NAME0_0_, this_.AGE as AGE0_0_
from CUSTOMERS this_
```

6．设置查询结果是否只允许读

Hibernate 从数据库中检索出来的对象都处于持久化状态，默认情况下，如果程序修改了持久化对象的属性，那么当 Hibernate 清理持久化缓存时，会通过相应的 SQL update 语句去同步更新数据库。

假如只允许读取持久化对象，而不允许更新它们，那么可以调用 Query 接口 setHint()方法来设置。值得注意的是，这里所说的不允许更新持久化对象，是指即使程序代码修改了内存中持久化对象的属性，当 Hibernate 清理持久化缓存时，也不会通过相应的 SQL update 语句去同步更新数据库。

以下程序代码演示如何把检索到的 Customer 对象设置为只允许读操作：

```
tx = entityManager.getTransaction();
tx.begin();

//返回的 Customer 对象处于持久化状态
Customer customer =entityManager
    .createQuery("from Customer c where c.id=1",Customer.class)
    //只允许读取检索到的持久化对象的属性
    //禁止更新检索到的持久化对象
    .setHint(org.hibernate.annotations.QueryHints.READ_ONLY,true)
    .getSingleResult();

//修改内存中 Customer 持久化对象的 name 属性
customer.setName("Mike");

//Session session=entityManager.unwrap(Session.class);
//session.setReadOnly(customer,false);

//修改内存中 Customer 持久化对象的 age 属性
customer.setAge(30);

//清理缓存，并提交事务。
tx.commit();
```

以上程序代码对检索到的Customer对象进行了修改属性的操作，但是当通过tx.commit()方法提交事务时，Hibernate 不会去同步更新数据库中 CUSTOMERS 表的相应记录。

Session 接口的 setReadOnly()方法可以设置单个持久化对象的读写模式：

- setReadOnly(object,true)：只允许读取持久化对象。
- setReadOnly(object,false)：允许读取或更新持久化对象。

如果把以上程序代码中的两行注释代码的注释取消掉：

```
Session session=entityManager.unwrap(Session.class);
session.setReadOnly(customer,false);  //允许更新 Customer 对象
```

那么当通过tx.commit()方法提交事务时，Hibernate会去同步更新数据库中CUSTOMERS表的相应记录，执行相应的 update 语句：

```
update CUSTOMERS set NAME='Mike',AGE=30 where ID=1;
```

17.1.14 定义命名查询语句

在前面的例子中，JPQL 查询语句都嵌入在程序代码中，这适用于比较简短的 JPQL 查询语句。如果 JPQL 查询语句很复杂，跨越多行，这会影响程序代码的可读性以及可维护性。

为了提高这些 JPQL 查询语句的可重用性，并提到程序代码的可读性和可维护性，可以把 JPQL 查询语句放在 JPA 或 Hibernate 的对象-关系映射文件中，也可以用特定的注解来定义。

一、在 JPA 的对象-关系映射文件中定义命名查询语句

以下例程 17-1 是 JPA 的对象-关系映射文件 orm.xml，它定义了两个命名查询语句：

- 采用 JPQL 查询语言，查询语句名为“findCustomersByName”。
- 采用本地 SQL 查询语言，查询语句名为“findCustomersSQL”。

例程 17-1　orm.xml

```
<entity-mappings
  version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
  http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">

  <named-query name="findCustomersByName">
    <query>
      <![CDATA[
        from Customer c where c.name like :name
       ]]>
     </query>
   </named-query>

  <named-native-query name="findCustomersSQL"
              result-class="mypack.Customer"  >
    <query>
        select * from CUSTOMERS
     </query>
   </named-native-query>
</entity-mappings>
```

在定义命名查询语句时，还可以通过<hint>元素来设置查询附属事项，例如：

```
  <named-query name="findCustomersByName">
    <query>
      <![CDATA[
        from Customer c where c.name like :name
       ]]>
     </query>

     <hint name="javax.persistence.query.timeout"
             value="60000" />
     <hint name="javax.persistence.query.comment"
             value="My comment ......" />

   </named-query>
```

二、在 Hibernate 的对象-关系映射文件中定义命名查询语句

以下例程 17-2 是 Hibernate 的对象-关系映射文件 Customer.hbm.xml，它定义为了两个命名查询语句：

- 采用 HQL 查询语言，查询语句名为“findCustomersByName”。
- 采用本地 SQL 查询语言，查询语句名为“findCustomersSQL”。

例程 17-2　Customer.hbm.xml

```
<hibernate-mapping >
  <class name="mypack.Customer" table="CUSTOMERS" >
   ……
  </class>

  <query name="findCustomersByName"
```

```
      cache-mode="get"   <!—设置对二级缓存的操作模式-->
      comment="My Comment..."   <!—设置注解-->
      fetch-size="50"   <!—设置批量抓取数目-->
      read-only="true"   <!—设置只读模式-->
      timeout="60">      <!—设置超时时间-->

    <![CDATA[
      from Customer c where c.name like :name
    ]]>
  </query>

  <sql-query name="findCustomersSQL">
    <return alias="c" class="mypack.Customer"/>
    select {c.*} from CUSTOMERS c where c.NAME like :name
  </sql-query>

</hibernate-mapping>
```

三、用注解定义命名查询语句

以下例程 17-3 是 Customer 类的部分代码，它用注解定义了两个命名查询语句：

- 采用来自 JPA API 的注解，查询语句名为 “findCustomersByName”。
- 采用来自 Hibernate API 的注解，查询语句名为 “findCustomersByAge”。

例程 17-3　Customer.java 的部分源代码

```
@NamedQueries({   //采用 JPA 的注解
  @NamedQuery(
    name="findCustomersByName",
    query="from Customer c where c.name like :name",
    //设置查询附属事项
    hints={
      @QueryHint(
        name=org.hibernate.annotations.QueryHints.TIMEOUT_JPA,
        value="60000"
      ),
      @QueryHint(
        name=org.hibernate.annotations.QueryHints.COMMENT,
        value="My Comment ......"
      )
    }
  )
})

@org.hibernate.annotations.NamedQueries({   //采用 Hibernate 的注解
  @org.hibernate.annotations.NamedQuery(
    name="findCustomersByAge",
    query="from Customer c where c.age > :age",
    timeout=60,
    comment="My Comment ......"
  )
})

@Entity
@Table(name="CUSTOMERS")
```

```
public class Customer  implements java.io.Serializable {……}
```

用 JPA API 中的注解来定义命名查询语句时，该定义代码只能放在持久化类中。而通过 Hibernate API 中的注解来定义命名查询语句时，该定义代码不仅能放在持久化类中，还可以放在包级别的类文件中。例如在 src/mypack/package-info.java 文件中定义如下命名查询语句：

```
@org.hibernate.annotations.NamedQueries({   //采用 Hibernate 的注解
  @org.hibernate.annotations.NamedQuery(
     name="findCustomersByAge",
     query="from Customer c where c.age > :age",
     timeout=60,
     comment="My Comment ......"
  )
})
package mypack;
```

四、用编程方式定义命名查询语句

在程序中，也可以先动态定义一个命名查询语句，名为“findCustomersByName”，在后的代码中，再执行这个命名查询语句：

```
//第一段程序代码
Query query =entityManager.createQuery(
                   "from Customer c where c.name like :name");
//把 Query 保存为命名查询语句
entityManager
   .getEntityManagerFactory()
   .addNamedQuery("findCustomersByName",query);

//以后 ……
//在其他程序代码中,获得并执行命名查询语句
TypedQuery<Customer> namedQuery=entityManager.createNamedQuery(
               "findCustomersByName",Customer.class);
List<Customer> result=namedQuery
                .setParameter("name","Tom")
                .getResultList();
```

五、获得并执行命名查询语句

通过以上的各种方式定义了命名查询语句后，在程序中需要使用该查询语句的地方，可以通过如下方式来获得并执行它：

```
//在其他程序代码中,获得并执行命名查询语句"findCustomersByName"
TypedQuery<Customer> namedQuery=entityManager.createNamedQuery(
               "findCustomersByName",Customer.class);
List<Customer> result=namedQuery
                .setParameter("name","Tom")
                .getResultList();
```

无论命名查询语句使用的是 JPQL 查询语言还是本地 SQL 查询语言，都可以通过 EntityManager 的 createNamedQuery()方法来获取。

如果命名查询语句中包含命名参数，则需要通过 Query 的 setParameter()方法来绑定参数。

17.1.15 调用函数

在 JPQL 中还可以调用函数，这些函数与 SQL 中的相应函数同名，作用也相同。例如

以下 JPQL 查询语句均调用了函数：

```
from Customer c where lower(c.name)='tom'

select upper(c.name) from Customer c

from Customer c where concat(c.firstname,c.lastname)
like ('G%  K%')

from Customer c where size(c.orders)>3  //客户的订单数目大于 3
```

在以上范例中，lower()函数把字符串中的所有字母转为小写；upper()函数把字符串中的所有字母转为大写；concat()函数把两个字符串连接起来；size()函数获取集合中元素的数目。

以下这段代码中的 JPQL 检索方式以及 QBC 检索方式都调用了 size()函数：

```
//JPQL 检索方式
String jpql= "from Customer c where size(c.orders)>3 ";
List<Customer> result= entityManager
          .createQuery(jpql,Customer.class)
          .getResultList();

//QBC 检索方式
CriteriaQuery<Customer> criteriaQuery =
     criteriaBuilder.createQuery( Customer.class );
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(root);

Predicate predicate=criteriaBuilder.gt(
      criteriaBuilder.size(root.get("orders")),3);
criteriaQuery.where(predicate);  //设置查询条件

List<Customer> result=entityManager
       .createQuery(criteriaQuery)
       .getResultList();
```

运行以上 Query 的 getResultList()方法时，Hibernate 执行的 select 语句为：

```
select * from CUSTOMERS  c
where (select count(o.CUSTOMER_ID) from ORDERS  o
      where c.ID=o.CUSTOMER_ID)>3
```

以下这段代码中的 JPQL 检索方式以及 QBC 检索方式都调用了 concat()函数：

```
//JPQL 检索方式
String jpql= "select concat('Name:',c.name) from Customer c";
List<String> result= entityManager
          .createQuery(jpql,String.class)
          .getResultList();

//QBC 检索方式
CriteriaQuery<String> criteriaQuery =
     criteriaBuilder.createQuery( String.class );
Root<Customer> root = criteriaQuery.from(Customer.class );
criteriaQuery.select(
    criteriaBuilder.concat("Name:",root.<String>get("name"));
```

```
List<String> result=entityManager
       .createQuery(criteriaQuery)
       .getResultList();
```

运行以上 Query 的 getResultList()方法时，Hibernate 执行的 select 语句为：

```
select concat('Name:',c.NAME) from CUSTOMERS  c;
```

JPA 提供的内置函数包括三类：字符串处理函数、算术函数和日期函数。字符串处理函数主要包括：

- concat(String s1, String s2)：字符串合并函数。
- substring(String s, int start, int length)：获得子字符串。例如 substring('HelloWorld',1,5)的返回值是“Hello”。
- trim(String s1)：从字符串中去掉首/尾指定的空格。
- lower(String s)：将字符串转换成小写形式。
- upper(String s)：将字符串转换成大写形式。
- length(String s)：求字符串的长度。
- locate(String s1, String s2, int start)：从第一个字符串中查找第二个字符串（子串）出现的位置。若未找到则返回 0。

算术函数主要包括：

- abs()：求绝对值。
- mod()：求余数。例如 mod(10,3)的返回值是 1，mod(10,2)的返回值是 0。
- sqrt()：求平方根。
- size()：求集合的元素个数。

日期函数主要包括以下三个函数，它们都不带参数：

- current_date()：返回服务器上的当前日期。
- current_time()：返回服务器上的当前时间。
- current_timestamp()：返回服务器上的当前时间戳。

在 JPQL 查询语句中可以使用以上所有的函数。CriterialBuilder 接口包含了和以上多数函数对应的同名方法，如 CriteriaBuilder.upper()、CriteriaBuilder.mod()等。例外情况是，对于日期函数，在 CriterialBuilder 接口中对应的方法是：currentDate()、currentTime()、currentTimestamp()。

JPQL 的一大灵活之处在于，假如在 JPQL 查询语句中调用了一个未在 Hibernate 中注册的函数，Hibernate 不会抛出错误，而是直接把这个函数交给底层数据库系统去执行。

17.1.16　元模型类（MetaModel Class）

本章 17.1.2 节已经讲过，QBC API 能够从元模型类中获得类的元数据信息。每个用注解映射的持久化类都对应一个元模型类。所有元模型类的类名以“_”结尾。例如 Customer 类的元模型类为 Customer_，以下例程 17-4 是本章的 Customer 类的元模型类的源程序。

例程 17-4　Customer_.java

```
package mypack;
import javax.annotation.Generated;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;
```

```
@Generated(value =
      "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Customer.class)
public abstract class Customer_ {
  public static volatile SingularAttribute<Customer, String> name;
  public static volatile SetAttribute<Customer, Order> orders;
  public static volatile SingularAttribute<Customer, Long> id;
  public static volatile SingularAttribute<Customer, Integer> age;

  public static final String NAME = "name";
  public static final String ORDERS = "orders";
  public static final String ID = "id";
  public static final String AGE = "age";
}
```

那么，元模型类是如何生成的呢？Hibernate 提供了元模型类生成器，它能够解析持久化类中的注解，生成相应的元模型类。

下面是用 Hibernate 为本章的 Customer 类和 Order 类生成元模型类的步骤。

（1）在 Hibernate 的安装软件包的 lib/jpa-metamodel-generator 目录下找到 hibernate-jpamodelgen-X.jar 类库文件，它包含了 Hibernate 的元模型类生成器。把这个类库文件拷贝到本章范例的 lib 目录下。

（2）到 Oracle 的官网上下载包含“javax.annotation”包的类库文件 jsr250-api.jar。把这个类库文件拷贝到本章范例的 lib 目录下。Hibernate 的元模型类生成器在生成元模型类时需要用到这个类库文件。

（3）接下来通过 ANT 工具编译本章范例时，就会为 Customer 类和 Order 类自动生成元模型类。Customer_类和 Order_类的.java 源文件以及.class 类文件都位于本章范例的 classes 目录下。

17.2 设定查询条件

和 SQL 查询语句一样，JPQL 查询语句也通过 where 子句来设定查询条件，例如：

```
from Customer c where c.name = 'Tom'
```

值得注意的是，在 where 子句中给出的是对象的属性名，而不是数据库表的字段名。

对于 QBC 查询，通过 CriteriaBuilder 的一系列方法来设定查询条件：

```
//指定查询条件
Predicate predicate = criteriaBuilder.gt(
                      root.get(Customer_.age), 21);
```

以上代码创建了一个简单的查询条件，用于比较 Customer 对象的 age 属性大于 21。

表 17-1 列出了 JPQL 和 QBC 在设定查询条件时可用的各种运算。

表 17-1 JPQL 和 QBC 支持的各种运算

运算类型	JPQL 运算符	QBC 运算方法	含义
比较运算	=	CriteriaBuilder.equal()	等于
	<>	CriteriaBuilder.notEqual()	不等于
	>	CriteriaBuilder.gt()	大于
	>=	CriteriaBuilder.ge()	大于等于

	<	CriteriaBuilder.lt()	小于
	<=	CriteriaBuilder.le()	小于等于
	is null	CriteriaBuilder.isNull()	等于空值
	is not null	CriteriaBuilder.isNotNull()	非空值
范围运算	in (列表)	CriteriaBuilder.in()	等于列表中的某一个值
	not in(列表)	CriteriaBuilder.not(CriteriaBuilder.in())	不等于列表中的任意一个值
	between 值 1 and 值 2	CriteriaBuilder.between()	大于等于值 1 并且小于等于值 2
	not between 值 1 and 值 2	CriteriaBuilder.not(CriteriaBuilder.between())	小于值 1 或者大于值 2
字符串模式匹配	like	CriteriaBuilder.like()	字符串模式匹配
逻辑运算	and	CriteriaBuilder.and()	逻辑与
	or	CriteriaBuilder.or()	逻辑或
	not	CriteriaBuilder.not()	逻辑非
集合运算	is empty	CriteriaBuilder.isEmpty()	集合为空，不包含任何元素
	is not empty	CriteriaBuilder.isNotEmpty()	集合不为空
	member of	CriteriaBuilder.isMember()	元素属于特定集合
	not member of	CriteriaBuilder.isNotMember()	元素不属于集合
数学运算	+，-	CriteriaBuilder.neg()（相当于负数符号）	正数和负数的符号
	+，-	CriteriaBuilder.sum() CriteriaBuilder.diff()	加法和减法运算
	*，/	CriteriaBuilder.prod() CriteriaBuilder.quot()	乘法和除法运算

在 JPQL 查询语句中，逻辑运算符用来连接多个查询条件，例如：

```
from Customer c  where (c.name like 'T%' and c.age>20 )
or c.email in ('xx@hotmail.com','yy@gmail.com')
```

在 JPQL 查询语句中还包括“.”、“+”、“-”、“*”和“/”等运算符，例如：

```
from Order o where (o.price/0.1)-100.0>0.0
```

以下表 17-2 列出了各种运算符在 JPQL 查询语句中的优先级。

表 17-2　各种运算符在 JPQL 查询语句中的优先级（从高到底）

运算符	描述
.	导航符
+，-	正数或者负数的符号
*，/	乘法/除法
+，-	加法/减法
=，<>，<，>，>=，<=，[not]between，[not]like，[not]in，is [not] null	比较运算符和范围运算符
is [not] empty，[not] member [of]	集合操作符
not，and，or	逻辑运算符

17.2.1　比较运算

下面举例说明如何在查询条件中进行比较运算。

（1）检索年龄不等于 18 的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery("from Customer c where c.age<>18 ");

//QBC 检索方式
Predicate predicate = criteriaBuilder.notEqual(root.get("age"), 18);
criteriaQuery.select(root).where(predicate);
```

（2）检索姓名为空的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery("from Customer c where c.name is null ");

//QBC 检索方式
Predicate predicate = criteriaBuilder.isNull(root.get("name"));
```

值得注意的是，不能通过以下 JPQL 查询语句来检索姓名为空的 Customer 对象：

```
from Customer c where c.name = null
```

和以上 JPQL 查询语句对应的 SQL 查询语句为：

```
select * from CUSTOMERS where NAME=null;
```

以上查询语句的查询结果永远为空，因为在 SQL 查询语句中，表达式（null=null）以及表达式（'Tom'=null）的比较结果既不是 true，也不是 false，而是 null。

（3）检索不属于任何客户的订单：

```
//JPQL 检索方式
entityManager.createQuery("from Order o where o.customer is null ");

//QBC 检索方式
CriteriaQuery<Order> criteriaQuery =
     criteriaBuilder.createQuery(Order.class);
Root<Order> root = criteriaQuery.from(Order.class);
Predicate predicate = criteriaBuilder.isNull(root.get("customer"));
```

（4）检索姓名为“Tom”的 Customer 对象，不区分大小写，如果 CUSTOMERS 表中记录的 NAME 字段值为“TOM”、“tOm”或者“tom”，都算满足查询条件的记录：

```
//JPQL 检索方式
Query query=entityManager.createQuery(
        "from Customer c where lower(c.name) ='tom' ");

//QBC 检索方式
Predicate predicate = criteriaBuilder.equal(
          criteriaBuilder.lower(root.get("name")),"tom");
```

或者：

```
//JPQL 检索方式
Query query=entityManager.createQuery(
        "from Customer c where upper(c.name) ='TOM' ");

//QBC 检索方式
Predicate predicate = criteriaBuilder.equal(
          criteriaBuilder.upper(root.get("name")),"TOM");
```

在 JPQL 查询语句或 QBC 查询中，都可以调用函数 lower()，它把字符串转为小写；或者调用 upper()函数，它把字符串转为大写。

（5）JPQL 查询和 QBC 查询都支持数学运算，例如：

```
//JPQL 检索方式
entityManager.createQuery(
        "from Order o where o.price/4-100>50 ");

//QBC 检索方式
Predicate predicate = criteriaBuilder.gt(
    criteriaBuilder.diff(
       criteriaBuilder.quot(root.<Double>get("price"),4)
     ,100)
 ,50);
```

17.2.2 范围运算

下面举例说明如何在查询条件中进行范围运算。

（1）检索姓名为 Tom、Mike 或者 Jack 的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
    "from Customer c where c.name in(' Tom ', ' Mike ', 'Jack ') ");

//QBC 检索方式
List<String> list=Arrays.asList(new String[]{"Tom","Mike","Jack"});
Predicate predicate =root.get("name").in(list);
```

以上 QBC 检索方式也可以用以下程序代码来实现：

```
CriteriaBuilder.In<String> inExpression =
                    criteriaBuilder.in(root.get("name"));
inExpression.value("Tom");
inExpression.value("Mike");
inExpression.value("Jack");
Predicate predicate=criteriaBuilder.and(inExpression);
```

（2）检索年龄在 18 到 25 之间的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
            "from Customer c where c.age between 18 and 25 ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.between(
      root.get("age"),18,25);
```

（3）检索年龄不在 18 到 25 之间的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
    "from Customer c where c.age not between 18 and 25 ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.not(
   criteriaBuilder.between(root.get("age"),18,25));
```

17.2.3 字符串模式匹配

和 SQL 查询一样，JPQL 用 like 关键字进行模糊查询，而 QBC 用 CriteriaBuilder 类的 like()方法进行模糊查询。模糊查询能够比较字符串是否与指定的字符串模式匹配，表 17-3 列出了字符串模式中可使用的通配符。

表 17-3 字符串模式中的通配符

通配符名称	通 配 符	作 用
百分号	%	匹配任意类型并且任意长度（长度可以为 0）的字符串，如果是中文，需要两个百分号，即“%%”
下划线	_	匹配单个任意字符，常用来限制字符串表达式的长度。

下面举例说明字符串模式匹配的用法。

（1）检索姓名以“T”开头的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
    "from Customer c where c.name like 'T%' ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.like(
    root.get("name"),"T%");
```

（2）检索姓名中包含字符串“om”的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
        "from Customer c where c.name like '%om%' ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.like(
    root.get("name"),"%om%");
```

（3）检索姓名以“T”开头，并且字符串长度为 3 的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
        "from Customer c where c.name like ' T_ _' ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.like(
    root.get("name")," T_ _");
```

17.2.4 逻辑运算

下面举例说明如何在查询条件中进行逻辑运算。

（1）检索姓名以“T”开头，并且以“m”结尾的 Customer 对象：

```
//JPQL 检索方式
Query query=entityManager.createQuery(
     "from Customer c where c.name"
     +"  like 'T%' and c.name like '%m' ");

//QBC 检索方式
Predicate predicate=criteriaBuilder.and(
   criteriaBuilder.like(root.get("name")," 'T%'"),
```

```
        criteriaBuilder.like(root.get("name")," '%m'"));
```

（2）检索姓名以“T”开头并且以“m”结尾，或者年龄不在 18 与 25 之间的 Customer 对象：

```
//JPQL 检索方式
Query query=entityManager.createQuery(
    "from Customer c "
    +"  where (c.name like 'T%' and c.name like '%m' )"
    +"or (c.age not between 18 and 25) " );

//QBC 检索方式
Predicate predicate=criteriaBuilder.or(
   criteriaBuilder.and(
     criteriaBuilder.like(root.get("name")," 'T%'"),
     criteriaBuilder.like(root.get("name")," '%m'")),
   criteriaBuilder.not(
     criteriaBuilder.between(root.get("name"),18,25)));
```

对比以上 JPQL 和 QBC 检索方式的程序代码，可以看出，如果查询条件非常复杂，QBC 检索会影响程序代码的可读性。

17.2.5 集合运算

下面举例说明如何在查询条件中进行集合运算。

（1）检索不包含任何订单的 Customer 对象：

```
//JPQL 检索方式
entityManager.createQuery(
      "from Customer c where c.orders is empty" );


//QBC 检索方式
Predicate predicate=criteriaBuilder
                  .isEmpty(root.get("orders"));
```

以上代码对应的 SQL select 语句为：

```
select ID,NAME,AGE from CUSTOMERS c
where not (exists
(select o.ID from ORDERS o where c.ID=o.CUSTOMER_ID))
```

（2）检索包含特定 Order 对象的 Customer 对象：

```
//先检索 ID=3 的 Order 对象
Order order=entityManager.find(Order.class,Long.valueOf(3));

//JPQL 检索方式
List<Customer> result=entityManager.createQuery("from Customer c"
       +" where :order member of c.orders",Customer.class )
      .setParameter("order",order)
      .getResultList();

//QBC 检索方式
CriteriaQuery<Customer> criteriaQuery =criteriaBuilder
                    .createQuery( Customer.class);
Root<Customer> root = criteriaQuery.from( Customer.class );
```

```
Predicate predicate=criteriaBuilder.isMember(
  (Expression<Order>)criteriaBuilder.parameter(Order.class,"order"),
  root.<Set<Order>>get("orders")
);

criteriaQuery.select(root)
             .where(predicate);

List<Customer> result=entityManager
       .createQuery(criteriaQuery)
       .setParameter("order",order)
       .getResultList();
```

对于以上两种检索方式，Query 的 getResultList()方法执行的 SQL select 语句为：

```
select * from CUSTOMERS c
where 3 in (select o.ID from ORDERS o where c.ID=o.CUSTOMER_ID)
```

17.2.6 case when 语句

在 JPQL 中可以使用“case when … else … end”多路分支语句，它的作用和 Java 语言的 switch 语句有点相似。例如在以下 JPQL 语句中，根据 Customer 类的 age 属性的大小返回相应的表示年龄状态的字符串：

```
select c.id,c.name,
         case
           when c.age>=18 then '成年人'
           when c.age>=14 then '青少年'
           else '儿童'
         end
from Customer c
```

以下程序代码执行上述 JPQL 语句，并且遍历访问查询结果：

```
String jpql="select c.id,c.name, "
       +"   case  when c.age>=18 then '成年人'  "
       +"   when c.age>=14 then '青少年' "
       +"   else '儿童'   end "
       +"   from Customer c";

TypedQuery<Object[]> query=
          entityManager.createQuery(jpql,Object[].class);
List<Object[]> result=query.getResultList();
Iterator<Object[]> it=result.iterator();
while(it.hasNext()){
  Object[] row=it.next();
  System.out.println("id:"+row[0]+",name:" +row[1]
       +",age status:"+row[2]);
}
```

如果 CUSTOMERS 表中所有的 AGE 字段都大于等于 18，那么以上代码将打印如下查询结果：

```
id:1,name:Tom,age status:成年人
id:2,name:Mike,age status:成年人
id:3,name:Jack,age status:成年人
```

```
id:4,name:Linda,age status:成年人
id:5,name:Tom,age status:成年人
```

17.3 小结

本章主要介绍了 JPQL 检索方式和 QBC 检索方式的基本用法，表 17-4 对此做了归纳。

表 17-4 JPQL 检索方式和 QBC 检索方式的基本用法

基本用法	范例	
使用别名	JPQL	from Customer **as c** where c.name=:name
	QBC	QBC 检索方式不需要由应用程序显式指定类的别名。
对查询结果排序	JPQL	Query query=entityManager.createQuery("from Customer c **order by c.name**");

	QBC	javax.persistence.criteria.Order order=
		criteriaBuilder.asc(root.get(Customer_.name));
		criteriaQuery.orderBy(order); //指定排序方式

分页查询	JPQL QBC	query. **setFirstResult(0)** **.setMaxResults(10);**
检索单个对象	JPQL QBC	Customer customer =(Customer)query**.setMaxResults(1)** **.getSingleResult();**
绑定参数	JPQL QBC	query.**setParameter("customerName",name)** **.setParameter("customerAge",age);**
设置查询附属事项	JPQL QBC	query.setFlushMode(FlushMode.COMMIT) .setHint("javax.persistence.cache.retrieveMode", CacheRetrieveMode.USE);
调用函数	JPQL	List<Customer> result= entityManager .createQuery("from Customer c where **size**(c.orders)>3" ,Customer.class)
	QBC	Predicate predicate=criteriaBuilder.gt(criteriaBuilder.**size**(root.get("orders")),3);
设定查询条件	JPQL	entityManager.createQuery("from Customer c where c.age<>18 ");
	QBC	Predicate predicate = criteriaBuilder.**notEqual**(root.get("age"), 18); criteriaQuery.select(root).where(predicate);

17.4 思考题

1．javax.persistence.Query 接口的哪些方法会执行 SQL select 语句？（多选）

（a）getResultList() （b）getSingleResult() （c）iterate() （d）setFirstResult()

2．在 QBC API 中，用于设定检索条件的 like()和 equal()方法位于哪个接口或类中？（单选）

（a）CriteriaQuery （b）CriteriaBuilder （c）Expression （d）Query

3．以下哪些程序代码能检索出按 name 属性升序排列的 Customer 对象？（多选）

（a）

```
List<Customer> result=entityManager
           .createQuery("from Customer c order by c.name",
            Customer.class)
           .getResultList();
```

（b）

```
CriteriaQuery<Customer> criteriaQuery =
```

```
        criteriaBuilder.createQuery(Customer.class);
    Root<Customer> root = criteriaQuery.from(Customer.class );
    criteriaQuery.select(root);

    javax.persistence.criteria.Order order=
      criteriaBuilder.asc(root.get(Customer_.name));
    criteriaQuery.orderBy(order);  //指定排序方式

    List<Customer> result=entityManger
                    .createQuery(criteriaQuery)
                    .getResultList();
```

（c）

```
    List<Customer> result=entityManager
      .createQuery("from Customer c order by c.name desc",Customer.class)
      .getResultList();
```

（d）

```
    List result=entityManager
       .createQuery("from Customer c order by c.name")
       .getResultList();
```

4．对于以下程序代码，以下哪些说法正确？（多选）

```
    EntityManager entityManager = entityManagerFactory
                          .createEntityManager();  //第 1 行
    EntityTransaction tx =
                         entityManager.getTransaction();  //第 2 行
    tx.begin();     //第 3 行

    Session session=entityManager.unwrap(Session.class);  //第 4 行

    Iterator<Customer> customers =session
       .createQuery("from Customer c  where c.age>10",Customer.class)
       .iterate();  //第 5 行

    while(customers.hasNext()){  //第 6 行
      Customer customer=customers.next(); //第 7 行
      System.out.println(customer.getName());  //第 8 行
    } //第 9 行

    tx.commit(); //第 10 行
    entityManager.close();  //第 11 行
```

（a）第 5 行程序代码执行的 SQL select 语句为：

```
    select ID from CUSTOMERS where AGE>10;
```

（b）执行完第 5 行程序代码，customers 集合中存放了所有 age 属性大于 10 的 Customer 代理类对象，它们的属性（除了 id 之外）没有被初始化。

（c）第 5 行程序代码执行的 SQL select 语句的形式为：

```
    select ID,NAME,AGE from CUSTOMERS where ID=?;
```

（d）第 7 行程序代码执行的 SQL select 语句的形式为：

```
    select ID,NAME,AGE from CUSTOMERS where ID=?;
```

5．ScrollableResults 接口的 previous()方法与以下哪个方法等价？（单选）
（a）setRowNumber(-1)　（b）first()　（c）scroll(-1)　（d）afterLast()

6．Query 接口的哪个方法用于设置清理 Session 的持久化缓存的模式？（单选）
（a）setCacheMode()　（b）setFetchMode()　（c）setLockMode()　（d）setFlushMode()

7．以下哪些程序代码能检索出不属于任何客户的订单？（多选）
（a）

```
List<Order> result=entityManager
            .createQuery("from Order o where o.customer is null ",
              Order.class)
            .getResultList();
```

（b）

```
CriteriaQuery<Order> criteriaQuery =
     criteriaBuilder.createQuery(Order.class);
Root<Order> root = criteriaQuery.from(Order.class);
Predicate predicate =
        criteriaBuilder.equal(root.get("customer"),"null");

criteriaQuery.select(root).where(predicate);
List<Order> result=entityManager.createQuery(criteriaQuery)
                                  .getResultList();
```

（c）

```
List<Order> result=entityManager
             .createQuery("from Order o where o.customer=null ",
               Order.class)
             .getResultList();
```

（d）

```
CriteriaQuery<Order> criteriaQuery =
     criteriaBuilder.createQuery(Order.class);
Root<Order> root = criteriaQuery.from(Order.class);
Predicate predicate = criteriaBuilder.isNull(root.get("customer"));

criteriaQuery.select(root).where(predicate);
List<Order> result=entityManager.createQuery(criteriaQuery)
                                  .getResultList();
```

8．假定数据库中 ID 为 1 的 CUSTOMERS 记录的 AGE 字段的值为 18。关于以下程序代码，哪些说法正确？（多选）

```
tx.begin();  //第 1 行
Customer customer =entityManger
    .createQuery("from Customer c where c.id=1",Customer.class)
    .setHint(
      org.hibernate.annotations.QueryHints.READ_ONLY,true)
     .getSingleResult();   //第 2 行

customer.setAge(customer.getAge()+1);  //第 3 行
tx.commit(); //第 4 行
```

（a）第 2 行执行 SQL select 语句。
（b）第 3 行执行一条 SQL update 语句，把数据库中 ID 为 1 的 CUSTOMERS 记录的 AGE 字段改为 19。

（c）第 4 行执行一条 SQL update 语句，把数据库中 ID 为 1 的 CUSTOMERS 记录的 AGE 字段改为 19。

（d）以上程序代码不会修改数据库中 ID 为 1 的 CUSTOMERS 记录的 AGE 字段。