第 14 章

利用HQL和JPA QL查询

本章内容

- 理解各种查询选项
- 编写HQL和JPA QL查询
- 联结、报表查询、子查询

查询是编写好的数据访问代码时最值得关注的部分。复杂的查询可能需要花很长的时间才能弄正确，因此它对应用程序的性能影响很巨大。另一方面，有了更多经验以后，编写查询就会变得更加容易，当初似乎很难的东西，只要了解了一些更高级特性，它们就不难解决了。

如果你已经使用手写的SQL多年，则可能关注ORM会拿掉一些你经常使用的表示法和灵活性。使用Hibernate和Java Persistence就没有这种问题。

Hibernate强大的查询工具允许你表达一般（或者甚至罕见地）在SQL中需要表达的几乎全部，但是从面向对象的角度——使用类和类的属性。

我们将介绍原生的Hibernate查询和Java Persistence中标准子集之间的区别。也可以把本章作为一个参考；因此有些小节写得不太详细，但是介绍了许多用于不同用例的小型代码示例。为了便于阅读，我们有时也在CaveatEmptor应用程序中跳过优化。例如，不引用`MonetaryAmount`值类型，而是在比较式中使用`BigDecimal`金额。

首先，介绍查询如何执行。别让自己被查询搞得心烦意乱；我们很快就会讨论到它们。

14.1 创建和运行查询

让我们从一些示例开始，以便你理解基础的用法。前面的章节提到过在Hibernate中有3种表达查询的方法：

- Hibernate查询语言（HQL），以及被标准化为JPA QL的子集：

```
session.createQuery("from Category c where c.name like 'Laptop%'");
entityManager.createQuery(
    "select c from Category c where c.name like 'Laptop%'"
);
```

- 用于按条件查询（QBC）和按示例查询（QBE）的`Criteria` API：

```
session.createCriteria(Category.class)
        .add( Restrictions.like("name", "Laptop%") );
```

- 直接的SQL（无论是否将结果集自动映射到对象）：

```
session.createSQLQuery(
    "select {c.*} from CATEGORY {c} where NAME like 'Laptop%'"
).addEntity("c", Category.class);
```

在执行之前，必须在应用程序代码中准备一个查询。因此，查询涉及几个独特的步骤：

(1) 创建查询，通过任何任意的限制或者你想要获取的数据的投影。

(2) 把运行时实参绑定到查询参数，该查询可以通过改变设置被重用。

(3) 执行针对数据库的预编译查询和数据获取。你可以控制查询如何执行，以及应该如何把数据获取到内存中（比如一次获取全部，还是逐个获取）。

Hibernate和Java Persistence在这些接口中提供查询接口和方法，以准备和执行任意的数据获取操作。

14.1.1 准备查询

org.hibernate.Query和org.hibernate.Criteria这两个接口都定义几种控制查询执行的方法。此外，Query还提供把具体值绑定到查询参数的方法。为了在应用程序中执行查询，需要利用Session获得其中一个接口的实例。

Java Persistence指定javax.persistence.Query接口。标准的接口不像原生的Hibernate API那么丰富，但是提供了以不同方式执行查询并把实参绑定到查询参数的所有必需的方法。不幸的是，有用的Hibernate Criteria API在Java Persistence中并没有等价物，虽然很可能在标准的未来版本中添加类似的查询接口。

1. 创建查询对象

为了创建新的Hibernate Query实例，在Session上调用createQuery()或者createSQLQuery()。createQuery()方法来准备一个HQL查询：

```
Query hqlQuery = session.createQuery("from User");
```

通过底层数据库的原生语法，可以用createSQLQuery()来创建SQL查询：

```
Query sqlQuery =
        session.createSQLQuery(
            "select {user.*} from USERS {user}"
        ).addEntity("user", User.class);
```

在这两个案例中，Hibernate都返回一个刚刚被实例化的Query对象，它可以用来指定一个特定的查询应该被如何执行，并允许执行查询。目前为止，还没有SQL被发送到数据库。

要想获得Criteria实例，就调用createCriteria()，传递你想让查询返回的对象的类。这也称作条件查询的根实体（root entity），如这个示例中的User：

```
Criteria crit = session.createCriteria(User.class);
```

Criteria实例的使用方法与Query对象一样——但它也用来构建查询的面向对象的表示法，通过添加Criterion实例并导航关联到新的Criteria。

利用JPA，你的查询起点是EntityManager。为了给JPA QL创建javax.persistence.Query实例，就调用createQuery()：

```
Query ejbQuery = em.createQuery("select u from User u");
```

为了创建原生的SQL查询，就用createNativeQuery()：

```
Query sqlQuery =
        em.createNativeQuery(
         "select u.USER_ID, u.FIRSTNAME, u.LASTNAME from USERS u",
         User.class
        );
```

你定义从原生查询返回对象的方法与Hibernate中的略有不同（这里的查询中没有占位符）。创建查询之后，要设置各种选项准备执行它。

2. 给结果进行分页

常用的方法是分页（pagination）。用户可以看到他们搜索请求（例如搜索特定的Item）的结果页面。这个页面一次显示有限的子集（假设10个Item），用户可以手工导航到后一个和前一个页面。在Hibernate中，Query和Criteria接口支持查询结果的这种分页：

```
Query query =
        session.createQuery("from User u order by u.name asc");
query.setMaxResults(10);
```

对setMaxResults(10)的调用把查询结果集限制为由数据库返回的前10个对象（行）。在这个Criteria查询中，被请求的页面从结果集的中间开始：

```
Criteria crit = session.createCriteria(User.class);
crit.addOrder( Order.asc("name") );
crit.setFirstResult(40);
crit.setMaxResults(20);
```

从第40个对象开始，获取接下来的20个对象。注意，在SQL中表达分页并没有标准的方法——Hibernate知道这些技巧，使得它能在特定的数据库中有效地工作。甚至可以把这个灵活的分页选项添加到一个SQL查询。Hibernate将给分页重写SQL：

```
Query sqlQuery =
        session.createSQLQuery("select {u.*} from USERS {u}")
                .addEntity("u", User.class);
sqlQuery.setFirstResult(40);
sqlQuery.setMaxResults(20);
```

可以通过Query和Criteria接口，使用方法链的编码风格（方法返回正接收的对象而不是void），重写前两个示例如下：

```
Query query =
        session.createQuery("from User u order by u.name asc")
                .setMaxResults(10);
Criteria crit =
        session.createCriteria(User.class)
                .addOrder( Order.asc("name") )
                .setFirstResult(40)
                .setMaxResults(20);
```

链方法调用没那么详细，许多Hibernate API都支持它。Java Persistence查询接口也通过javax.persistence.Query接口，对JPA QL和原生的SQL查询支持分页和方法链：

```
Query query =
        em.createQuery("select u from User u order by u.name asc")
          .setFirstResult(40)
          .setMaxResults(20);
```

准备查询的下一步是设置任何运行时参数。

3. 考虑参数绑定

如果没有运行时参数绑定，你只能编写很糟糕的代码：

```
String queryString =
        "from Item i where i.description like '" + search + "'";
List result = session.createQuery(queryString).list();
```

你应该永远不要编写这样的代码，因为恶意的用户可能搜索下面的项目描述——也就是说，在搜索对话框中输入search的值为：

```
foo' and callSomeStoredProcedure() and 'bar' = 'bar
```

可见，原始的queryString不再是对字符串的一个简单查询，而且还在数据库中执行一个存储过程！引号没有进行转码，因此在查询中对存储过程的调用是另一个有效的表达式。如果你编写了一个像这样的查询，允许在你的数据库中执行任意的代码，就是在应用程序中开了一个很大的安全漏洞。这就是众所周知的SQL注入（SQL injection）安全性问题。永远不要将用户输入的未被检查值传递到数据库！幸而，有一种简单的机制可以防止这种错误。

JDBC驱动包括将值安全绑定到SQL参数的功能。它准确地知道参数值中的哪些字符要转码，因此不存在前面提到的安全性问题。例如，对给定的search中的引号字符进行转码，不再作为控制符号处理，而是作为搜索字符串值的一部分。此外，使用参数的时候，数据库能够有效地高速缓存预编译的语句，显著地改善性能。

参数绑定（parameter binding）有两种方法：利用定位或者利用具名参数。Hibernate和Java Persistence都支持这两个选项，但是你无法给一个特定的查询同时使用这两个选项。

利用具名参数，可以重写这个查询为：

```
String queryString =
    "from Item item where item.description like :search";
```

参数名称后面的冒号表示这是一个具名参数。然后，对search参数绑定一个值：

```
Query q = session.createQuery(queryString)
                 .setString("search", searchString);
```

由于searchString是一个用户提供的字符串变量，你调用Query接口的setString()方法，把它绑定到具名参数（:search）。这个代码更规则、更安全且执行得更好，因为如果只是绑定参数的变化，单独编译过的SQL语句就可以被重用。

你将经常需要多个参数：

```
String queryString = "from Item item"
                   + " where item.description like :search"
                   + " and item.date > :minDate";
```

```
Query q = session.createQuery(queryString)
                 .setString("search", searchString)
                 .setDate("minDate", mDate);
```

同样的查询和代码，在Java Persistence中看起来稍有不同：

```
Query q = em.createQuery(queryString)
            .setParameter("search", searchString)
            .setParameter("minDate", mDate, TemporalType.DATE);
```

setParameter()方法是个一般的操作，它可以绑定实参的所有类型，对于临时的类型只需要一点点帮助(引擎需要知道你是只要日期、时间，还是需要完整的时间戳绑定)。Java Persistence只支持这种方法，用于参数的绑定（顺便说一下，Hibernate也有这种方法)。

另一方面，Hibernate还提供许多其他的方法，其中一部分是为了完整，其他的是为了方便，你可以用它们来把实参绑定到查询参数。

4．利用Hibernate的参数绑定

你已经调用了setString()和setDate()来把实参绑定到查询参数。原生的Hibernate Query接口提供类似的便利方法，用来绑定大多数Hibernate内建类型的参数：从setInteger()到setTimestamp()和setLocale()的一切。它们中大多数是可选的；可以依赖setParameter()方法自动找出正确的类型（除了临时的类型之外)。

特别有用的一种方法是setEntity()，它让你绑定一个持久化实体(注意，setParameter()很聪明，足以自动理解这一点)：

```
session.createQuery("from Item item where item.seller = :seller")
       .setEntity("seller", theSeller);
```

然而，还有一种允许绑定任何Hibernate类型实参的一般方法：

```
String queryString = "from Item item"
                   + " where item.seller = :seller and"
                   + " item.description like :desc";

session.createQuery(queryString)
       .setParameter( "seller",
                      theSeller,
                      Hibernate.entity(User.class) )
       .setParameter( "desc", description, Hibernate.STRING );
```

这甚至适用于定制的用户自定义的类型，如MonetaryAmount：

```
Query q = session.createQuery("from Bid where amount > :amount");
q.setParameter( "amount", givenAmount,
                Hibernate.custom(MonetaryAmountUserType.class) );
```

如果你有包含seller和description属性的JavaBean，就可以调用setProperties()方法来绑定查询参数。例如，可以在Item类自身的实例中传递查询参数：

```
Item item = new Item();
item.setSeller(seller);
item.setDescription(description);

String queryString = "from Item item"
```

```
                    + " where item.seller = :seller and"
                    + " item.description like :desccription";

session.createQuery(queryString).setProperties(item);
```

setProperties()绑定使得JavaBean属性的名称与查询字符串中的具名参数一致，内部调用setParameter()来猜测Hibernate类型并绑定值。实际上，结果表明它并没有听起来得这么有用，因为有些常用的Hibernate类型是不可猜测的（尤其是临时的类型）。

Query的参数绑定方法是可以为空（null-safe）的。因此下列代码是有效的：

```
session.createQuery("from User as u where u.username = :name")
        .setString("name", null);
```

然而，这段代码的结果几乎确定不是你想要的！生成的SQL将包含一个比较式（如USERNAME=null，它在SQL三重逻辑中始终取值为空）。反之，你必须使用is null操作符：

```
session.createQuery("from User as u where u.username is null");
```

5. 利用定位参数

如果喜欢，可以在Hibernate和Java Persistence中用定位参数代替：

```
String queryString = "from Item item"
                    + " where item.description like ?"
                    + " and item.date > ?";

Query q = session.createQuery(queryString)
                 .setString(0, searchString)
                 .setDate(1, minDate);
```

Java Persistence也支持定位参数：

```
String queryString = "from Item item"
                    + " where item.description like ?1"
                    + " and item.date > ?2";

Query q = em.createQuery(queryString)
                .setParameter(1, searchString)
                .setParameter(2, minDate, TemporalType.DATE);
```

这段代码不仅比使用具名参数的方法更难懂，而且如果稍微改变查询字符串，还有更容易破碎的弱点：

```
String queryString = "from Item item"
                    + " where item.date > ?"
                    + " and item.description like ?";
```

绑定参数位置的每一个变化都需要改变参数绑定代码。这样就产生了易碎和更需要维护的代码。建议避免使用定位参数。如果你通过编程构建复杂的查询，它可能更加方便，但是，此时Criteria API是更好的选择。

如果必须使用定位参数，要记住Hibernate是从0开始计数，而Java Persistence则是从1开始计数，必须给JPA QL查询字符串中的每一个问号都添加数字。它们有着不同的遗留根：Hibernate的根是JDBC，Java Persistence的根是更早版本的EJB QL。

除了绑定参数之外，你还会经常想要应用其他影响查询如何执行的提示（hint）。

6. 设置查询提示

假设你在执行查询之前修改了持久化对象。这些修改只出现在内存中，因此Hibernate（和Java Persistence提供程序）在执行查询之前，把持久化上下文和所有的变化清除到数据库。这样保证了查询在当前的数据中运行，并保证在查询结果和内存之间不会出现冲突。

这有时候是不现实的：例如，如果你执行一系列包含许多个查询－修改－查询－修改的操作，并且每次查询都获取一个与以前不同的数据集。换句话说，不需要在执行查询之前把修改清除到数据库，因为冲突的结果不是问题。注意，持久化上下文给实体对象提供可重复读取，因此无论如何都只有查询的标量结果才是问题。

可以在Session或者EntityManager中，利用setFlushMode()禁用持久化上下文的清除。或者，如果你想要只在特定的查询之前禁用清除，可以在Query（Hibernate和JPA）对象中设置一个FlushMode：

```
Query q = session.createQuery(queryString)
                 .setFlushMode(FlushMode.COMMIT);
Query q = em.createQuery(queryString)
            .setFlushMode(FlushModeType.COMMIT);
```

Hibernate不会在执行任何这些查询之前清除持久化上下文。

另一种优化是对特定查询结果的一个细粒度的org.hibernate.CacheMode。你在13.4.5节中用了一种高速缓存模式，控制Hibernate如何与二级高速缓存交互。如果Hibernate通过标识符获取对象，它在一级持久化上下文高速缓存中查找，如果二级高速缓存被启用，还在二级高速缓存区域查找这个实体。执行一个返回实体实例的查询时，也发生同样的事：在查询结果的封送期间，Hibernate试图先从持久化上下文高速缓存中查找它们，来解析所有的实体实例——如果实体实例处在持久化上下文高速缓存中，它就忽略查询结果的实体数据。并且，如果获取到的实体实例不在任何高速缓存中，Hibernate就会在查询结束之后把它放在高速缓存中。可以在查询中通过CacheMode控制这个行为：

```
Query q = session.createQuery("from Item")
                 .setCacheMode(CacheMode.IGNORE);

Criteria criteria = session.createCriteria(Item.class)
                           .setCacheMode(CacheMode.IGNORE);

Query q = em.createQuery(queryString)
            .setHint("org.hibernate.cacheMode",
                     org.hibernate.CacheMode.IGNORE);
```

例如，CacheMode.IGNORE告诉Hibernate不要为这个查询返回的任何实体而与二级高速缓存交互。换句话说，通过这个查询获取到的任何Item都不放在二级高速缓存中。如果你执行一个不应该更新二级高速缓存的查询，设置这种高速缓存模式就很有用，或许因为你正在获取的数据只与特定的情况相关，因此不应该耗尽高速缓存区域中的可用空间。

9.3.3节讲过持久化上下文的控制，以及如何减少内存消耗并防止很长的脏查询周期。给特定的持久化对象禁用脏查询的一种方法是设置session.setReadOnly(object,true)(EntityMa-

nager不支持这个API)。

你可以告诉Hibernate：由查询返回的所有实体对象都应该被当作是只读的（虽然不是脱管）：

```
Query q = session.createQuery("from Item")
                 .setReadOnly(true);
Query q = em.createQuery("select i from Item i")
              .setHint("org.hibernate.readOnly", true);
```

由这个查询返回的所有Item对象都处于持久化状态，但是在持久化上下文中，没有给自动脏检查启用任何快照。Hibernate不会自动持久化任何修改，除非用session.setReadOnly(object, false)禁用只读模式。

可以通过设置超时（timeout），来控制允许一个查询运行多久：

```
Query q = session.createQuery("from Item")
                 .setTimeout(60); // 1 minute

Criteria criteria = session.createCriteria(Item.class)
                           .setTimeout(60);
Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.timeout", 60);
```

这种方法与JDBC Statement中的setQueryTimeout()方法有着相同的语义和结果。而且还与底层JDBC相关的是抓取大小（fetch size）：

```
Query q = session.createQuery("from Item")
                 .setFetchSize(50);

Criteria criteria = session.createCriteria(Item.class)
                           .setFetchSize(50);

Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.fetchSize", 50);
```

JDBC抓取大小是对数据库驱动程序的一个优化提示；如果驱动程序没有实现这个功能，它就不能导致任何性能改善。如果它实现了，当客户端在一个查询结果（即ResultSet）中操作时，通过在一个批量中获取许多个行，可以改善JDBC客户端和数据库之间的通信。由于Hibernate幕后正在使用ResultSet，如果用list()执行一个查询，这个提示就可以改善数据获取——我们很快会实践这一点。

优化应用程序的时候，经常必须阅读复杂的SQL日志。强烈建议启用hibernate.use_sql_comments；然后Hibernate就会为它写到日志中的每条SQL语句添加一条注释。可以通过setComment()给特定的查询设置定制的注释：

```
Query q = session.createQuery("from Item")
                 .setComment("My Comment...");

Criteria criteria = session.createCriteria(Item.class)
                           .setComment("My Comment...");

Query q = em.createQuery("select i from Item i")
            .setHint("org.hibernate.comment", "My Comment...");
```

目前为止你已经设置的提示全部都与Hibernate或者JDBC处理相关。许多开发人员（和数据

库管理员）把查询提示当作是完全不同的东西。在SQL中，查询提示是SQL语句中的一条注释，它给数据库管理系统的SQL优化器包含了一条指令。例如，如果开发人员或者数据库管理员认为，数据库优化器为特定的SQL语句所选择的执行计划不是最快的，他们就用提示来强制一个不同的执行计划。Hibernate和Java Persistence的API不支持任意的SQL提示，必须退回到原生的SQL并编写自己的SQL语句——当然，你可以通过所提供的API来执行语句。

（通过一些数据库管理系统，可以在SQL语句一开始时就通过SQL注释来控制优化器。在这个例子中，是用Query.setComment()添加提示。在其他的场景中，可以编写org.hibernate.Interceptor，并在它被发送到数据库之前，在onPrepareStatement(sql)方法中操作SQL语句。）

最后，可以控制查询是否应该在数据库管理系统中强制悲观锁——这是一直持续到数据库事务结束的锁：

```
Query q = session.createQuery("from Item item")
                 .setLockMode("item", LockMode.UPGRADE);

Criteria criteria = session.createCriteria(Item.class)
                           .setLockMode(LockMode.UPGRADE);
```

这两个查询，如果得到数据库方言的支持，都会生成一个包括... FOR UPDATE操作（或者其等价物，如果得到数据库系统和方言的支持的话）的SQL声明。目前，悲观锁在Java Persistence查询接口中不可用（但它被规划为一个Hibernate扩展提示）。

如果现在准备好了查询，那么你可以运行它们了。

14.1.2　执行查询

一旦你创建并准备好了Query或者Criteria对象，就准备执行它，并把结果获取到内存中去。在一次循环中把整个结果获取到内存中去，这是执行查询的一种最常用的方法，我们称之为列表（listing）。接下来我们也要讨论一些其他的可用选项：迭代（iterating）和滚动（scrolling）。滚动的用处与迭代差不多：你很少需要这些选项。我们猜测在常规的应用程序中，有90%的查询执行都依赖list()和getResultList()方法。

首先来看一个最常见的案例。

1．列出所有结果

在Hibernate中，list()方法执行查询并返回结果为java.util.List：

```
List result = myQuery.list();
```

Criteria接口也支持这个操作：

```
List result = myCriteria.list();
```

在这两个案例中，是立即执行一个还是几个SELECT语句，这取决于你的抓取计划。如果映射任何关联或者集合为非延迟，就必须抓取它们，另外还要抓取你想要通过查询获取的数据。所有这些对象都被加载到内存中，获取到的任何实体对象都处于持久化状态，并被添加到持久化上下文。

Java Persistence用相同的语义提供了一种方法，但是用不同的名称：

```
List result = myJPAQuery.getResultList();
```

你知道，利用有些结果只是单个实例的查询——例如，如果你只想要最高的出价。如果是这样，就可以利用索引result.get(0)从结果清单中读取它。或者可以利用setMaxResult(1)限制返回的行数。然后，可以通过uniqueResult()方法执行查询，因为你知道只会返回一个对象：

```
Bid maxBid =
    (Bid) session.createQuery("from Bid b order by b.amount desc")
                 .setMaxResults(1)
                 .uniqueResult();
Bid bid = (Bid) session.createCriteria(Bid.class)
                       .add( Restrictions.eq("id", id) )
                       .uniqueResult();
```

如果查询返回不止一个对象，就抛出异常。如果查询结果为空，就返回null。这种情况在Java Persistence中也适用，只是方法的名称不同罢了（不幸的是，如果结果为空，就会抛出异常）：

```
Bid maxBid = (Bid) em.createQuery(
                       "select b from Bid b order by b.amount desc"
                     ).setMaxResults(1)
                     .getSingleResult();
```

把所有的结果获取到内存中是执行查询最常用的方法。Hibernate支持一些其他的方法，如果你想要优化查询的内存消耗和执行行为，可能会发现它们很有帮助。

2. 循环访问结果

Hibernate Query接口也提供iterate()方法来执行查询。它返回与list()一样的数据，但是依赖不同的策略来获取结果。

调用iterate()执行查询时，Hibernate在第一个SQL SELECT中只获取实体对象的主键（标识符）值，然后试图在持久化上下文高速缓存和（如果启用）二级高速缓存中查找对象的其他状态。

考虑下列代码：

```
Query categoryByName =
    session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
List categories = categoryByName.list();
```

这个查询导致至少一个SQL SELECT的执行，CATEGORY表的所有列都包括在该SELECT子句中：

```
select CATEGORY_ID, NAME, PARENT_ID from CATEGORY where NAME like ?
```

如果你希望类别已经被高速缓存在持久化上下文或者二级高速缓存中，那就只需要标识符值（高速缓存的键）。这样减少了从数据库中抓取的数据量。下列SQL效率更高一些：

```
select CATEGORY_ID from CATEGORY where NAME like ?
```

可以给它使用iterate()方法：

```
Query categoryByName =
    session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
Iterator categories = categoryByName.iterate();
```

初始的查询只获取Category主键值。然后你循环访问结果，Hibernate在当前的持久化上下文和二级高速缓存（如果启用的话）中查找每一个Category对象。如果高速缓存没有被访问到，Hibernate会给每次循环都执行一个额外的SELECT，通过Category对象的主键，从数据库中完整地获取它。

在大多数情况下，这是一项很小的优化。把行读取减到最少通常比把列读取减到最少更为重要。尽管如此，如果你的对象有很大的字符串字段，这种方法也可能有助于把网络中的数据包减到最少，因此把延迟也减到最小。应该清楚，只有当被迭代实体的二级高速缓存区域被启用时这种方法才真正有效。否则，它就会产生n+1查询问题！

Hibernate保持迭代器开着，直到你循环访问了所有的结果，或者直到Session关闭。也可以通过org.hibernate.Hibernate.close(iterator)显式地关闭它。

还要注意，在编写本书之时，Hibernate Criteria和Java Persistence还不支持这一优化。

执行查询的另一种优化方法是翻阅结果。

3．利用数据库游标滚动

简单的JDBC提供一种特性，称作可滚动的结果集（scrollable resultset）。这种方法使用一种保存在数据库管理系统中的游标。游标指向查询结果中一个特定的行，应用程序可以前后移动游标。甚至可以通过游标跳到一个特定的行。

应该翻阅查询结果（而不是把它们全部加载到内存中）的其中一种情形是结果集太大而无法载入内存。通常可以尝试通过增加查询中的条件进一步限制结果。但有时候这是不可能的，或许因为你需要所有的数据，却又想要分几步获取。

12.2.2节中已介绍过滚动，也讲解了如何实现处理批量数据的过程，因为这是它最能大展身手的地方。下列示例概述了ScrollableResults接口中其他值得关注的选项：

```
ScrollableResults itemCursor =
    session.createQuery("from Item").scroll();

itemCursor.first();
itemCursor.last();
itemCursor.get();

itemCursor.next();
itemCursor.scroll(3);
itemCursor.getRowNumber();
itemCursor.setRowNumber(5);
itemCursor.previous();
itemCursor.scroll(-3);

itemCursor.close();
```

这段代码没有太大的意义；它显示了ScrollableResults接口中最值得关注的方法。可以把游标设置到结果中的第一个和最后一个Item对象，或者使用获得游标当前正使用指向的Item。可以通过setRowNumber()跳到一个位置，到达一个特定的Item，或者通过previous()和next()前后滚动。另一种方法是利用scroll()，通过偏移前后滚动。

执行Hibernate Criteria查询也可以通过滚动而不是list()，返回的ScrollableResults游标的工作原理相同。注意，在终止数据库事务之前、用完游标之后绝对必须关闭它。下面是

Criteria示例，展现了游标的打开：

```
ScrollableResults itemCursor =
    session.createCriteria(Item.class)
            .scroll(ScrollMode.FORWARD_ONLY);

... // Scroll only forward

itemCursor.close()
```

Hibernate API的ScrollMode常量相当于简单JDBC中的常量。在这个示例中，该常量确保游标只能向前移动。这是必需的，以防万一，因为有些JDBC驱动程序并不支持向后滚动。其他可用的方法是ScrollMode.SCROLL_INSENSITIVE和ScrollMode.SCROLL_SENSITIVE。在游标开着时，迟钝的游标不会把你公开给被修改的数据（有效地保证没有脏读取、不可重复读取、或者幻读可以滑入到结果集中）。另一方面，当你在结果集中工作时，灵敏的游标会把刚刚被提交的数据和修改公开给你。注意，Hibernate持久化上下文高速缓存仍然给实体实例提供可重复读取，因此只有你在结果集中投影的被修改的标量值会受到这项设置的影响。

目前为止，我们介绍的代码示例全部都在Java代码中嵌入了查询字符串文字。对于简单的查询来说，这样做不无道理，但是一旦开始考虑必须被分成多行的复杂查询时，这就变得有点难以处理了。

14.1.3 使用具名查询

我们不想在Java代码中看见到处分散着HQL或者JPA QL字符串文字，除非真正有必要。Hibernate让你把查询字符串具体化到映射元数据，这种技术称作*具名查询*（named query）。它让你把与特定的持久化类或者一组类（该类封装了它的其他元数据）相关的所有查询都保存在一个XML映射文件中。或者，如果使用注解，就可以创建具名查询作为特定实体类的元数据，或者把它们放在一个XML部署描述符中。查询的名称被用来从应用程序代码中进行调用。

1. 调用具名查询

在Hibernate中，getNamedQuery()方法给具名的查询获得Query实例：

```
session.getNamedQuery("findItemsByDescription")
        .setString("desc", description);
```

在这个例子中，你调用了具名查询findItemsByDescription，并把一个字符串参数绑定到具名参数desc。

Java Persistence也支持具名查询：

```
em.createNamedQuery("findItemsByDescription")
   .setParameter("desc", description);
```

具名查询是全局的——也就是说，查询的名称被当作是特定的SessionFactory或者持久化单元的唯一标识符。它们如何以及在哪里定义，在XML映射文件中还是在注解中，都与应用程序代码无关。甚至查询语言也不关心。

2. 在XML元数据中定义具名查询

可以把一个具名查询放在XML元数据中任何<hibernate-mapping>元素内部。在更大的应

用程序中，建议把所有具名查询隔离并分离到它们自己的文件中去。或者，你可能想让某些查询在相同的XML映射文件中定义为一个特定的类。

<query>定义具名的HQL或者JPA QL查询：

```
<query name="findItemsByDescription"><![CDATA[
        from Item item where item.description like :desc
]]></query>
```

应该把查询文本包装在一个CDATA指令中，以便XML解析器不会被查询字符串中可能意外地被当作XML的任何字符而混淆（例如 < 操作符）。

如果你把具名查询定义放在<class>元素内部，而不是在根的内部，它就用实体类的名称作为前缀。例如，findItemsByDescription也可以称作auction.model.Item.findItemsByDescription。否则，你要确保查询的名称是全局唯一的。

你之前利用API设置的所有查询提示也可以通过声明设置：

```
<query name="findItemsByDescription"
       cache-mode="ignore"
       comment="My Comment..."
       fetch-size="50"
       read-only="true"
       timeout="60"><![CDATA[
    from Item item where item.description like :desc
]]></query>
```

具名查询不一定是HQL或者JPA QL字符串；它们甚至可以是原生的SQL查询——并且你的Java代码也不需要知道这种差别：

```
<sql-query name="findItemsByDescription">
    <return alias="item" class="Item"/>
    <![CDATA[
        select {item.*} from item where description like :desc
    ]]>
</sql-query>
```

如果你认为稍后可能想要通过调优SQL来优化查询，这就很有用。如果必须把一个遗留应用程序移植到Hibernate，其中SQL代码从手工编码的JDBC子程序中隔离开来，这也是一种很好的解决方案。利用具名查询，可以轻松地逐个地把查询移植到映射文件。第15章将更详细地讨论原生的SQL查询。

3. 利用注解定义具名查询

Java Persistence标准指定@NameQuery和@NamedNativeQuery注解。可以把这些注解放进特定类的元数据内，也可以放在JPA XML描述符文件内。注意，查询名称在任何情况下都必须是全局唯一的；不会自动加上任何类或者包名称作为前缀。

假设你考虑一个特定的具名查询属于特定的实体类：

```
package auction.model;

import ...;

@NamedQueries({
    @NamedQuery(
```

```
        name = "findItemsByDescription",
        query = "select i from Item i where i.description like :desc"
      ),
      ...
})
@Entity
@Table(name = "ITEM")
public class Item { ... }
```

一种更为常用的解决方案就是把查询封装在orm.xml部署描述符中：

```
<entity-mappings ...>
    ...
    <named-query name="findAllItems">
        <query>select i from Item i</query>
    </named-query>

    <entity class="Item">
        ...
        <named-query name="findItemsByDescription">
          <query>
               select i from Item i where i.description like :desc
          </query>
          <hint name="org.hibernate.comment" value="My Comment"/>
          <hint name="org.hibernate.fetchSize" value="50"/>
          <hint name="org.hibernate.readOnly" value="true"/>
          <hint name="org.hibernate.timeout" value="60"/>
        </named-query>
    </entity>

</entity-mappings>
```

你会看到Java Persistence描述符支持扩展点：named-query定义的hint元素。可以用它设置特定于Hibernate的提示，就像之前用Query接口通过编程进行的那样。

原生的SQL查询有它们自己的元素，也可以在一个实体映射的内部或者外部定义它们：

```
<named-native-query name="findItemsByDescription"
                    result-set-mapping="myItemResult">
    <query>select i.NAME from ITEM i where i.DESC = :desc</query>
    <hint name="org.hibernate.timeout" value="200"/>
</named-native-query>
```

嵌入原生的SQL比我们目前为止介绍过的都更强大（可以定义任意的结果集映射）。第15章将回来讨论其他的SQL嵌入选项。

是否要利用具名查询特性，这要由你自己决定。然而，我们把应用程序代码中的查询字符串（除非它们处在注解中）当作次选；如有可能，你通常应该把查询字符串具体化到外部的映射文件中去。

你现在知道如何通过Hibernate、JPA以及元数据创建、准备和执行查询了。下面应该更深入地学习查询语言和选项。我们从HQL和JPA QL开始。

14.2 基本的 HQL 和 JPA QL 查询

从一些简单的查询开始，来熟悉HQL语法和语义。我们应用选择来指定数据源，用限制使记

录与条件相匹配，并用投影选择想要从查询中返回的数据。

试一试　测试Hibernate查询——用于Eclipse IDE的Hibernate Tools支持Hibernate Console（控制台）视图。可以在控制台窗口测试查询，并立即看到生成的SQL和结果。

你还将在本节中学习JPA QL，因为它是HQL功能的一个子集——必要时我们会提到其中的区别。

本节谈到查询时，通常是指SELECT语句，即从数据库中获取数据的操作。HQL也支持UPDATE、DELETE甚至INSERT .. SELECT语句，就如12.2.1节所述。这里不再重复这些大批量操作，而是关注SELECT语句。但是记住，HQL和JPA QL之间的某些区别也可能适用于大批量操作——例如，一个特定的函数是否可移植。

HQL中的SELECT语句甚至不用SELECT子句就可以生效；只需要FROM。JPA QL则不是这样，在JPA QL中SELECT子句不是可选的。这实际上没有太大的区别；几乎所有的查询都需要SELECT子句，无论你用JPA QL还是HQL编写。然而，我们要从FROM子句开始探讨查询，因为依据我们的经验，这样更容易理解。记住，要把这些查询变成JPA QL，理论上你必须添加SELECT子句来完成语句，但是如果你忘了（假设是SELECT *），无论如何Hibernate还是会让你执行查询。

14.2.1　选择

HQL中最简单的查询是单个持久化类的一个选择（注意，这里我们不是指SELECT子句或者语句，而是指从数据被选择的位置）：

```
from Item
```

这个查询生成下列SQL：

```
select i.ITEM_ID, i.NAME, i.DESCRIPTION, ... from ITEM i
```

1. 利用别名

通常，通过使用HQL或者JPA QL选择一个类进行查询时，你需要分配一个别名给被查询的类，用作查询其他部分的一个引用：

```
from Item as item
```

as关键字通常是可选的。上面的代码相当于下列代码：

```
from Item item
```

把这个稍微想象成下列Java代码中的临时变量声明：

```
for ( Iterator i = allQueriedItems.iterator(); i.hasNext(); ) {
    Item item = (Item) i.next();
    ...
}
```

把别名item分配到Item类的被查询实例，允许你稍后在代码（或者查询）中引用它们的属性值。为了提醒自己这个类似之处，建议你给别名使用与临时变量[通常采用骆驼命名法（camelCase）]相同的命名约定。然而，本书中的某些示例可能使用更短的别名，例如用i代替

item，以保证印刷出来的代码易于阅读。

常见问题 HQL和JPA QL区分大小写吗？我们从来不用大写字母编写HQL和JPA QL关键字，也从来不用大写字母编写SQL关键字。那样看起来难看并且古板——最现代的终端机可以显示大写和小写字符。然而，HQL和JPQ QL对于关键字并不区分大小写，因此如果你喜欢，也可以编写为FROM Item AS item。

2. 多态查询

作为面向对象的查询语言，HQL和JPA QL支持多态查询（polymorphic query）——相应地查询类的实例和它子类的所有实例。你已经掌握了足够的HQL和JPA QL知识来证明这一点。考虑下列查询：

```
from BillingDetails
```

这句代码返回类型BillingDetails的对象，它是个抽象类。在这个例子中，具体的对象属于BillingDetails:CreditCard和BankAccount的子类型。如果只想要一个特定子类的实例，可以使用：

```
from CreditCard
```

在from子句中具名的类甚至不需要一定是被映射的持久化类，任何类都可以！下列查询返回所有的持久化对象：

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

当然，这也适用于接口——这个查询返回所有可序列化的持久化对象：

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

同样地，下列条件查询返回所有的持久化对象（是的，可以通过这样一个查询来选择数据库的所有表）：

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

注意，Java Persistence并没有标准化利用未被映射的接口的多态查询。然而，这适用于Hibernate EntityManager。

多态不仅仅应用到FROM子句中显式命名的类，还应用到多态关联，就像你将在本章稍后会见到的那样。

已经讨论过了FROM子句，现在进入HQL和JPA QL的其他部分。

14.2.2 限制

通常，你并不想要获取一个类的所有实例。你必须能够在查询返回的对象的属性值中表达约束。这称作限制。WHERE子句用来表达SQL、HQL和JPA QL中的限制。这些表达式可能跟你正在寻找的需要缩小的代码块一样复杂。注意，限制不仅应用到SELECT语句，还可用于限定UPDATE或者DELETE操作的范围。

这是一个典型的WHERE子句，它限制结果为带有指定的电子邮件地址的所有User对象：

```
from User u where u.email = 'foo@hibernate.org'
```

注意，该约束是根据User类的属性email进行表达的，并给它用了一个面向对象的概念。这个查询生成的SQL是：

```
select u.USER_ID, u.FIRSTNAME, u.LASTNAME, u.USERNAME, u.EMAIL
 from USER u
 where u.EMAIL = 'foo@hibernate.org'
```

可以在语句和条件中用单引号包括文字。HQL和JPA QL中常用的其他文字是TRUE和FALSE：

```
from Item i where i.isActive = true
```

限制利用三重逻辑进行表达。WHERE子句是一个逻辑表达式，给对象的每一个字节组（tuple）都取值为true、false或者null。利用内建的比较操作符，通过把对象的属性与其他属性或者文字值进行对比，来构建逻辑表达式。

常见问题　什么是三重逻辑？当且仅当WHERE子句取值为true时，行才被包括在SQL结果集中。在 Java 中，notNullObject==null 取值为 false，null==null 取值为 true。在 SQL 中，NOT_NULL_COLUMN=null和null=null都取值为null，而不是true。这样，SQL就需要一个特定的操作符IS NULL来测试一个值是否为null。这个三重逻辑是处理表达式的一种方法，可以应用到空的列值。不要把null当作特殊的标记，而是作为普通的值处理，这是对关系模型中常见的二重逻辑的SQL扩展。HQL和JPA QL必须利用三重操作符支持这个三重逻辑。

让我们快速看一下最常用的比较操作符。

1. 比较表达式

HQL和JPA QL支持与SQL相同的基本比较操作符。如果了解SQL，以下是一些看起来应该很熟悉的示例：

```
from Bid bid where bid.amount between 1 and 10
from Bid bid where bid.amount > 100
from User u where u.email in ('foo@bar', 'bar@foo')
```

因为底层的数据库实现了三重逻辑，测试null值需要很谨慎。记住null = null在SQL中不会取值为true，而是为null。使用null操作数的所有比较都取值为null。（这就是为什么通常在查询中看不到null文字的原因。）HQL和JPA QL提供SQL风格的IS [NOT] NULL操作符：

```
from User u where u.email is null
from Item i where i.successfulBid is not null
```

这个查询返回所有没有电子邮件地址的用户和被出售的货品。

LIKE操作符允许通配符搜索，在这里通配符符号是%和_，就像SQL中的一样：

```
from User u where u.firstname like 'G%'
```

这个表达式限制了结果为带有firstname的以大写字母G开头的用户。也可以否定LIKE操作符，例如在一个子字符串匹配表达式中：

```
from User u where u.firstname not like '%Foo B%'
```

百分比符号代表字符的任何一种顺序；下划线可以用来通配单个字符。如果想要文字百分比或者下划线，可以定义转义符：

```
from User u where u.firstname like '\%Foo%' escape='\'
```

这个查询返回带有名字的以%Foo开头的所有用户。

HQL和JPA QL支持算术表达式：

```
from Bid bid where ( bid.amount / 0.71 ) - 100.0 > 0.0
```

逻辑操作符（和用于分组的圆括号）用于组合表达式：

```
from User user
        where user.firstname like 'G%' and user.lastname like 'K%'
from User u
        where ( u.firstname like 'G%' and u.lastname like 'K%' )
        or u.email in ('foo@hibernate.org', 'bar@hibernate.org' )
```

可以在表14-1中从上到下看到操作符的优先顺序。

表14-1 HQL和JPA QL操作符优先顺序

操 作 符	描 述
.	导航路径表达式操作符
+,-	一元的正号或负号（所有无正负之分的数值均被当作正值）
*,/	数值的一般乘法和除法
+,-	数值的一般加法和减法
=, <>, <, >, >=, <=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OR]	包含SQL语义的二元比较操作符
NOT, AND, OR	HQL和JPA QL中用于集合的二元操作符 用来排列表达式取值的逻辑操作符

这里列出的操作符和它们的优先顺序在HQL和JPA QL中是一样的。算术操作符（例如乘法和加法）都是一目了然的。你已经见过二重比较表达式如何有着与它们的SQL配对物相同的语义，以及如何通过逻辑操作符对它们进行分组和组合。我们来讨论集合处理。

2. 包含集合的表达式

前几节中的所有表达式都只包括单值的路径表达式：user.email、bid.amount等。也可以通过正确的操作符，在查询的WHERE子句中使用以集合结束的路径表达式。

例如，假设你想要通过集合的大小来限制查询结果：

```
from Item i where i.bids is not empty
```

这个查询返回它们的bids集合中有元素的所有Item实例。也可以表达你需要一个特定的元素出现在一个集合中：

```
from Item i, Category c where i.id = '123' and i member of c.items
```

这个查询返回Item和Category实例——通常添加SELECT子句，并只投影这两个实体类型的其中之一。它返回包含主键'123'（单引号中的一个文字）的Item实例，以及与这个Item实例关

联的所有Category实例。（这里使用的另一个技巧是特殊的.id路径；这个字段始终指向一个实体的数据库标识符，无论标识符属性的名称是什么。）

在HQL和JPA QL中使用集合还有许多其他的方法。例如，可以在函数调用中使用它们。

3. 调用函数

HQL一项极为强大的特性是在WHERE子句中调用SQL函数的能力。如果数据库支持用户自定义的函数（大多数都可以），不管怎样，你就可以把这一点用于各种目的。至于现在，我们要考虑使用标准的ANSI SQL函数UPPER()和LOWER()。这些可以用于区分大小写的搜索：

```
from User u where lower(u.email) = 'foo@hibernate.org'
```

另一个常用的表达式是拼接——虽然这里的SQL方言不同，但HQL和JPA QL支持可移植的concat()函数：

```
from User user
        where concat(user.firstname, user.lastname) like 'G% K%'
```

同样典型的是需要集合大小的表达式：

```
from Item i where size(i.bids) > 3
```

JPA QL标准化了最常用的函数，如表14-2中所概括的。

表14-2　标准的JPA QL函数

函　数	适　用　性
UPPER(s), LOWER(s)	字符串值。返回一个字符串值
CONCAT(s1, s2)	字符串值。返回一个字符串值
SUBSTRING(s, offset, length)	字符串值（从1开始偏移）。返回一个字符串值
TRIM([[BOTH|LEADING|TRAILING] char [FROM]] s)	如果没有指定char或者其他的规范，就去除s的两端空白。返回一个字符串值
LENGTH(s)	字符串值。返回一个数值
LOCATE(search, s, offset)	从offset开始搜索s字符串中search字符串所在的位置。返回一个数字值
ABS(n), SQRT(n), MOD(dividend, divisor)	数字值。返回一个与输入类型相同的绝对值，double类型的平方根，以及一个整数除以另一个整数之后产生的integer类型的余数
SIZE(c)	集合表达式。返回一个integer，或者如果为空则返回0

所有标准的JPA QL函数都可以用在查询的WHERE和HAVING子句中（稍后你很快会见到）。原生的HQL更灵活一点。首先，它提供其他的可移植函数，如表14-3所示。

表14-3　其他的HQL函数

函　数	适　用　性
BIT_LENGTH(s)	返回s的位数
CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP()	返回数据库管理系统机器的日期和（或）时间
SECOND(d), MINUTE(d), HOUR(d), DAY(d), MONTH(d), YEAR(d)	从临时的实参中提取时间和日期

（续）

函 数	适 用 性
CAST(t as Type)	把指定的类型t转换为Hibernate的Type
INDEX(joinedCollection)	返回被联结的集合元素的索引
MINELEMENT(c), MAXELEMENT(c), MININDEX(c), MAXINDEX(c), ELEMENTS(c), INDICES(c)	返回被索引集合（映射、列表、数组）的元素或者索引
已经在org.hibernate.Dialect中注册的	以一种方言扩展包含其他函数的HQL

这些HQL函数中的大多数，转变成了可能你之前已经用过的SQL中的配对物。这个转换表可以通过org.hibernate.Dialect定制和扩展。检查你正用给数据库的方言的源代码；你可能在那里发现有许多已经注册的其他SQL函数，可以在HQL中立即使用。记住，没有包括在org.hibernate.Dialect超类中的每一个函数，对于其他的数据库管理系统都是不可移植的！

另一个新增到Hibernate API的是Hibernate Configuration API中的addSqlFunction()方法：

```
Configuration cfg = new Configuration();
cfg.addSqlFunction(
     "lpad",
     new StandardSQLFunction("lpad", Hibernate.STRING)
);
... cfg.buildSessionFactory();
```

这个操作把SQL函数lpad添加到HQL。请见StandardSQLFunction及其子类的Javadoc，了解更多的信息。

当你调用一个没有对SQL方言进行注册的函数时，HQL甚至尝试变得聪明些：在HQL语句的WHERE子句中调用的且不为Hibernate所知的任何函数，被作为SQL函数调用直接传递到数据库。如果你不在乎数据库的可移植性，这种方法就很棒，否则就必须注意那些不可移植的函数。

最后，在探讨HQL和JPA QL中的SELECT子句之前，看看结果是如何排序的。

4. 排序查询结果

所有的查询语言都提供一些排序查询结果的机制。HQL和JPA QL提供了ORDER BY子句（类似于SQL）。

这个查询返回所有用户，按用户名排序：

```
from User u order by u.username
```

利用asc或者desc指定按升序或者降序排序：

```
from User u order by u.username desc
```

可以按多种属性进行排序：

```
from User u order by u.lastname asc, u.firstname asc
```

你现在知道如何编写FROM、WHERE和ORDER BY子句了。你知道如何选择想要获取的实体的实例，以及选择限制和排序结果所需的表达式和操作。你现在需要的就是把这个结果的数据投影到你在应用程序中需要的东西的能力。

14.2.3　投影

SELECT子句在HQL和JPA QL中执行投影。它允许你准确指定需要查询结果中的哪些对象或者对象的哪些属性。

1．实体和标量值的简单投影

例如，考虑下列HQL查询：

```
from Item i, Bid b
```

这是有效的HQL查询，但在JPA QL中是无效的——标准要求你使用SELECT子句。尽管如此，Item和Bid的乘积所隐含的结果，同样也可以通过显式的SELECT子句生成。这个查询返回有序的Item和Bid实例对：

```
Query q = session.createQuery("from Item i, Bid b");
// Query q = em.createQuery("select i, b from Item i, Bid b");

Iterator pairs = q.list().iterator();
// Iterator pairs = q.getResultList().iterator();

while ( pairs.hasNext() ) {
    Object[] pair = (Object[]) pairs.next();
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

这个查询返回Object[]的一个List。索引0处为Item，索引1处为Bid。因为这是一个乘积，结果包含了在两张底层的表中找到的Item和Bid行的每一种可能的组合。很显然，这个查询没用，但是当你收到一个Object[]的集合作为查询结果时，也不应该感到惊讶。

下列显式的SELECT子句也返回Object[]的一个集合：

```
select i.id, i.description, i.initialPrice
    from Item i where i.endDate > current_date()
```

这个查询返回的Object[]在索引0处包含了一个Long，在索引1处包含一个String，并在索引2处包含一个BigDecimal或者MonetaryAmount。这些都是标量值，而不是实体实例。因此，它们不处于任何持久化状态，就像实体实例那样。它们不是事务的，显然也不会自动对脏状态进行检查。我们称这种查询为标量查询（scalar query）。

2．获取独特的结果

当你使用SELECT子句时，结果的元素不再保证是唯一的。例如，货品描述就不是唯一的，因此下列查询不止一次地返回相同的描述：

```
select item.description from Item item
```

很难理解在一个查询结果中有两个相同的行会有什么意义，因此如果你认为可能有重复，一般就使用DISTINCT关键字：

```
select distinct item.description from Item item
```

这样就从返回的Item描述列表中消除了重复。

3．调用函数

（对于某些Hibernate SQL方言来说）也可能从SELECT子句中调用数据库特定的SQL函数。例

如，下列查询从数据库服务器（Oracle语法）中获取当前的日期和时间，以及Item的一个属性：

```
select item.startDate, current_date() from Item item
```

SELECT子句中数据库函数的技术不受限于依赖数据库的函数。它也适用于其他更多一般的（或者标准的）SQL函数：

```
select item.startDate, item.endDate, upper(item.name)
    from Item item
```

这个查询返回Object[]，包含一件货品拍卖的起始和终止日期，货品的名称全部大写。

在实践中，也可能调用SQL统计函数（aggregate function），本章稍后会讨论到。然而，注意Java Persistence标准和JPA QL不保证任何非统计函数的函数可以在SELECT子句中调用。Hibernate和HQL提供更多的灵活性，因此我们认为支持JPA QL的其他产品将在某种程度上提供同样的自主。还要注意，Hibernate不知道的函数并没有作为SQL函数调用被传递到数据库，就像在WHERE子句中一样。你必须在org.hibernate.Dialect中注册一个函数，在HQL中的SELECT子句中启用它。

前几节应该已经让你从基本的HQL和JPA QL开始了。现在该看一下更复杂的查询选项了，例如联结、动态抓取、子查询和报表查询。

14.3 联结、报表查询和子查询

很难把有些查询归类为是高级的，而把其他查询归类为基本的。很显然，我们在本章前几节中介绍过的查询还没能让你做到这一点。

至少你还需要知道联结是如何工作的。任意地联结数据的能力是关系数据访问的基础能力之一。联结数据也是基本的操作，它使你能够在单个查询中抓取几个被关联的对象和集合。我们现在要介绍基本的联结操作如何工作，以及如何使用它们编写动态抓取策略。

我们认为属于高级的其他技术，包括通过有效地对结果进行统计和分组的子查询和报表查询等各种语句的嵌套。

让我们从联结以及如何用联结进行动态抓取开始。

14.3.1 联结关系和关联

用一个联结组合两个（或者更多个）关系中的数据。例如，可以联结ITEM和BID表中的数据，如图14-1所示。（注意，并非所有的列和可能的行都显示出来；因此用省略号表示。）

ITEM

ITEM_ID	DESCRIPTION	...
1	Item Nr. One	...
2	Item Nr. Two	...
3	Item Nr. Three	...

BID

BID_ID	ITEM_ID	AMOUNT
1	1	99.00
2	1	100.00
3	1	101.00
4	2	4.99

图14-1 ITEM和BID表是联结操作明显的备选表

当人们在SQL数据库的上下文中听到联结一词时，大都认为是内部联结。内部联结是几种联结类型中最重要的一种，并且最容易理解。考虑SQL语句及图14-2。这个SQL语句是FROM子句中的ANSI类型的内部联结。

```
select i.*, b.* from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
```

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99

图14-2　两张表的ANSI类型内部联结的结果表

如果你通过内部联结来联结表ITEM和BID，就利用它们的一般属性（ITEM_ID列），在一个新的结果表中获得所有货品和它们的出价。注意，这个操作的结果只包含有出价的货品。如果想要所有的货品，并且某种货品没有相应的出价时，就用NULL值代替其出价数据，此时就可以利用(左)外部联结，如图14-3所示。

```
select i.*, b.* from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID
```

ITEM_ID	DESCRIPTION	...	BID_ID	ITEM_ID	AMOUNT
1	Item Nr. One	...	1	1	99.00
1	Item Nr. One	...	2	1	100.00
1	Item Nr. One	...	3	1	101.00
2	Item Nr. Two	...	4	2	4.99
3	Item Nr. Three	...	NULL	NULL	NULL

图14-3　两张表的ANSI类型左外部联结的结果

你可以认为表联结像下面这样工作。首先，通过取ITEM行与BID行所有可能的组合，得到两张表的一个乘积。

第二，使用联结条件过滤这些被联结的行。（任何一个好的数据库引擎都有较复杂的算法来计算联结。它通常不会构建消耗内存的乘积，然后过滤所有的行。）联结条件是一个布尔表达式，如果被联结的行会被包括在结果中，就取值为真。对于左外部联结而言，从不满足联结条件的(左)ITEM表中的每一行也都包括在结果中，给BID的所有列返回NULL值。

*右*外部联结获取所有的出价，或null（如果一个出价没有对应的货品）——此时这不是一个有意义的查询。右外部联结很少使用；开发人员始终考虑从左到右，并且先放置驱动表。

在SQL中，联结条件通常被显式指定。（不幸的是，不可能使用外键约束的名称来指定两张表如何联结。）你为ANSI类型的联结在ON子句中（或者为所谓的theta类型的联结在WHERE子句中）指定联结条件：`where I.ITEM_ID=B.ITEM_ID`。

我们现在讨论HQL和JPA QL联结选项。记住这两者都是基于SQL并转变成SQL，因此，即使语法稍有不同，你也应该始终参考前面介绍过的两个示例，并验证一下你所理解的结果SQL和结果集看起来是什么样子。

1. HQL和JPA QL联结选项

在Hibernate查询中，通常不显式指定联结条件，而是指定被映射的Java类关联的名称。这基本上与我们在SQL中想要的特性一样，通过一个外键约束名称表达联结条件。由于你已经在Hibernate中映射了大多数（如果不是全部）数据库Schema的外键关系，因此可以在查询语言中使用这些被映射关联的名称。这是真正的语法糖（syntactical sugar）[①]，但很方便。

例如，Item类有一个包含Bid类的关联具名bids。如果你在一个查询中命名这个关联，Hibernate在映射文档中就有足够的信息，然后推断出表联结表达式。这有助于使查询更为简短、更易于阅读。

事实上，HQL和JPA QL提供了4种表达（内部和外部）联结的方法：

- 隐式关联联结；
- FROM子句中的普通联结；
- FROM子句中的抓取联结；
- WHERE子句中theta类型的联结。

稍后介绍如何编写两个没有定义关联的类之间的（一个theta类型）联结，以及如何在查询的FROM子句中编写普通联结和抓取联结。

隐式关联联结是常用的缩写。（注意，我们决定要通过通常省略SELECT子句，使接下来的示例更易于阅读和理解——在HQL中有效，在JPA QL中则无效。）

2. 隐式关联联结

目前为止，你已经使用过简单的限定属性名，比如查询中的bid.amount和item.description。HQL和JPA QL支持大部分属性路径表达式，用一个点号表示两种不同的目的：

- 查询组件；
- 表达隐式关联联结。

第一种用法很简单：

```
from User u where u.homeAddress.city = 'Bangkok'
```

你用一个点号引用被映射组件Address的几个部分。在这个查询中没有联结任何表，homeAddress组件的属性与User数据一起全部被映射到相同的表。也可以在SELECT子句中编写一个路径表达式：

```
select distinct u.homeAddress.city from User u
```

这个查询返回String的一个List。由于重复没有太大的意义，因此用DISTINCT把它们排除。

大部分路径表达式的第二种用法是隐式关联联结：

```
from Bid bid where bid.item.description like '%Foo%'
```

① 指在不改变语法含义的前提下，把语法变得更加易于阅读。——编者注

这样就在从Bid到Item的多对一关联上产生了一个隐式联结——这个关联的名称是item。Hibernate知道你用BID表中的ITEM_ID外键映射了这个关联，并相应地生成SQL联结条件。隐式联结始终采用多对一或者一对一关联，从来不通过集合值的关联(无法编写item.bids.amount)。

单个路径表达式中可能存在多个联结。如果从Item到Category的关联是多对一（而不是目前的多对多），就可以这样编写：

```
from Bid bid where bid.item.category.name like 'Laptop%'
```

我们反对给更复杂的查询使用这种语法糖。SQL联结很重要，尤其当优化查询的时候，你需要能够一眼就看到有多少个联结。考虑下列查询（再次利用从Item到Category的多对一关联）：

```
from Bid bid
    where bid.item.category.name like 'Laptop%'
    and bid.item.successfulBid.amount > 100
```

在SQL中表达这个需要多少个联结？即使找到了正确答案，也花了你不少时间。答案是3个，生成的SQL看起来有点像这样：

```
select ...
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join CATEGORY C on I.CATEGORY_ID = C.CATEGORY_ID
inner join BID SB on I.SUCCESSFUL_BID_ID = SB.BID_ID
where C.NAME like 'Laptop%'
and SB.AMOUNT > 100
```

如果在FROM子句中用显式的HQL和JPA QL联结来表达这个查询，它会更明显。

3. 在FROM子句中表达的联结

Hibernate区分不同的联结目的。假设你正在查询Item，你可能想要使用Bid来联结它们的原因有两种。

第一，你可能想要以应该被应用到货品Bid的一些条件为基础，来限制通过查询返回的货品。例如，你可能想要出价超过$100的所有Item；因而这需要一个内部联结。目前你还不关注没有出价的货品。

另一方面，你可能主要关注Item，但可能想要执行一个外部联结，只是因为想给在同一个SQL语句中查询的Item获取所有Bid，类似于我们之前称作即时联结抓取的东西。记住，你宁可在默认情况下延迟映射所有的关联，因此，即时的外部联结抓取查询，用于在运行时给特定的用例覆盖默认的抓取策略。

先来编写一些使用内部联结进行限制的查询。如果想要获取Item实例，并限制结果为其出价具有特定金额的货品，就必须为被联结的关联分配一个别名：

```
from Item i
    join i.bids b
        where i.description like '%Foo%'
        and b.amount > 100
```

这个查询为实体Item分配了别名i，为被联结的Item出价分配了别名b。然后可以用这两个别名在WHERE子句中表达限制条件。

生成的SQL是：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
       b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
and b.AMOUNT > 100
```

查询以有序的配对形式返回被关联的Bid和Item的所有组合：

```
Query q = session.createQuery("from Item i join i.bids b");
Iterator pairs = q.list().iterator();
while ( pairs.hasNext() ) {
    Object[] pair = (Object[]) pairs.next();
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

这个查询没有返回Item的一个List，而是返回Object[]数组的一个List。在索引0处为Item，索引1处为Bid。一个特定的Item可能出现多次，每个被关联的Bid一次。这些重复的货品是重复的内存引用，而不是重复的实例！

如果你不想要查询结果中的Bid，可以在HQL中指定一个SELECT子句（无论如何，它对于JPA QL都是强制的）。在SELECT子句中使用别名，只投影想要的对象：

```
select i
from Item i join i.bids b
where i.description like '%Foo%'
and b.amount > 100
```

现在，生成的SQL看起来像这样：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
from ITEM i
inner join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
and b.AMOUNT > 100
```

查询结果只包含Item，由于它是一个内部联结，因此只有包含Bid的Item：

```
Query q = session.createQuery("select i from Item i join i.bids b");
Iterator items = q.list().iterator();
while ( items.hasNext() ) {
    Item item = (Item) items.next();
}
```

如你所见，在HQL和JPA QL中使用别名，对于直接的类和被联结的关联都是一样的。你在前面的示例中用了一个集合，但是对于单值的关联来说，其语法和语义都是相同的，例如多对一和一对一。通过命名关联，你在FROM子句中分配别名，然后在WHERE（或许也在SELECT子句）中使用别名。

HQL和JPA QL提供另一种语法，用于在FROM子句中联结集合，并给它分配一个别名。这个IN()操作符在更早版本的EJB QL中就出现了。它的语义与普通集合联结的一样。可以重写最后一个查询如下：

```
select i
from Item i in(i.bids) b
where i.description like '%Foo%'
and b.amount > 100
```

from Item i in(i.bids) b生成了与之前的示例中用from Item i join i.bids b一样的内部联结。

目前为止，你还只是编写了内部联结。外部联结主要用于动态抓取，我们很快会讨论到。有时候有你想要用外部联结而不应用动态的抓取策略，来编写一个简单的查询。例如，下列查询是第一种查询的变形，它获取金额最小的货品和出价：

```
from Item i
    left join i.bids b
        with b.amount > 100
        where i.description like '%Foo%'
```

这个语句中的第一个新事物是LEFT关键字。可以选择编写LEFT OUTER JOIN和RIGHT OUTER JOIN，但是我们通常更喜欢简短的形式。第二个变化是WITH关键字后面的额外联结条件。如果把b.amount > 100表达式放进WHERE子句中，你就可以限制结果为有出价的Item实例。但这并不是你在此处想要的：你想要获取货品和出价，甚至没有出价的货品。通过在FROM子句中添加额外的联结条件，可以限制Bid实例并且仍然获取所有的Item对象。这个查询再次返回有序的Item和Bid对象对。最后，注意包含WITH关键字的额外联结条件只在HQL中可用；JPA QL只支持由被映射的外键关联所表示的基本的外部联结条件。

外部联结在其中起着重要作用的一种更常见的场景是即时动态抓取。

4．利用联结的动态抓取策略

你在前一节见过的所有查询都有一个共同点：返回的Item实例都有一个名为bids的集合。如果映射为lazy="true"（默认），这个集合就没有被初始化，一旦你访问它，就立即触发一个额外的SQL语句。对于所有单端的关联也是一样，就像每个Item的seller。默认情况下，Hibernate生成一个代理，并只按需延迟加载被关联的User实例。

有哪些选项可以改变这个行为呢？首先，可以在映射元数据中改变抓取计划，并声明一个集合或者单值的关联为lazy="false"。然后，Hibernate执行必要的SQL，保证始终加载想要的对象网络。这也意味着单个HQL或者JPA QL语句可能产生几个SQL操作！

另一方面，通常不在映射元数据中修改抓取计划，除非绝对确定它应该全局应用。你通常给一个特定的用例编写新的抓取计划。这也是你通过编写HQL和JPA QL语句所已经做过的事；通过选择、限制和投影定义了一个抓取计划。使它更有效的唯一东西是正确的动态抓取策略（dynamic fetching strategy）。例如，你没必要用几个SQL语句来抓取所有的Item实例并初始化它们的bids集合，或者给每个Item获取seller。这可以通过一个联结操作同时完成。

在HQL和JPA QL中，可以指定被关联的实体实例或者集合应该在FROM子句中通过FETCH关键字即时抓取：

```
from Item i
    left join fetch i.bids
        where i.description like '%Foo%'
```

这个查询返回描述中包含字符串“Foo”的所有货品，以及它们在单个SQL操作中的所有出价集合。执行时，它返回Item实例的一个列表，其中它们的bids集合已被完全初始化。如果把它与前一节中的由查询返回的有序对进行对比，就会发现它完全不同！

抓取联结的目的在于性能优化：你使用这个语法，只是因为想要在单个SQL操作中即时初始化bids集合：

```
select i.DESCRIPTION, i.INITIAL_PRICE, ...
       b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
from ITEM i
left outer join BID b on i.ITEM_ID = b.ITEM_ID
where i.DESCRIPTION like '%Foo%'
```

额外的WITH子句在这里没有太大的意义。无法限制Bid实例：所有的集合都必须被初始化。也可以利用相同的语法预抓取多对一或者一对一关联：

```
from Bid bid
    left join fetch bid.item
    left join fetch bid.bidder
        where bid.amount > 100
```

这个查询执行下列SQL：

```
select b.BID_ID, b.AMOUNT, b.ITEM_ID, b.CREATED_ON
       i.DESCRIPTION, i.INITIAL_PRICE, ...
       u.USERNAME, u.FIRSTNAME, u.LASTNAME, ...
from BID b
left outer join ITEM i on i.ITEM_ID = b.ITEM_ID
left outer join USER u on u.USER_ID = b.BIDDER_ID
where b.AMOUNT > 100
```

如果你编写JOIN FETCH，没用LEFT，那么就通过一个内部联结获得了即时加载（如果使用INNER JOIN FETCH也一样）；例如，利用内部联结的预抓取，返回其bids集合完全初始化的Item对象，不返回没有出价的Item对象。这样的查询很少用于集合，但是可以用于不可为空的多对一关联。例如，join fetch item.seller的效果就很好。

HQL和JPA QL中的动态抓取很简单，但是你应该牢记下列警告：

- 永远不要为任何被抓取联结的关联或者集合分配别名，以便进一步限制或者投影。因此left join fetch i.bids b where b =...是无效的，而left join fetch i.bids b join fetch b.bidder却是有效的。
- 不应该并行抓取一个以上的集合；否则就产生了笛卡儿积。你喜欢抓取几个单值的关联对象就可以抓取几个，而不会产生乘积。这基本上与我们在13.2.5节中讨论过的问题一样。
- HQL和JPA QL忽略你已经在映射元数据中定义的任何抓取策略。例如，在XML中通过fetch="join"映射bids集合，对任何HQL或者JPA QL语句都没有影响。动态的抓取策略忽略全局的抓取策略（另一方面，不会忽略全局的抓取计划——保证加载每一个非延迟的关联或者集合，即使需要几个SQL查询）。
- 如果即时抓取一个集合，则可能返回重复。看看图14-3：这就是给select i from Item i join fetch i.bids HQL或者JPA QL查询执行的那个SQL操作。每个Item都在结果表的左侧重复，重复次数与相关的Bid数据出现的次数一样。HQL或者JPA QL查询返回的

List把这些重复保存为引用。如果喜欢过滤掉这些重复，需要把List包装在一个Set中（例如，用Set noDupes =new LinkedHashSet(resultList)，或者利用DISTINCT关键字：select distinct i from Item i join fetch i.bids——注意，在这个例子中，DISTINCT不在SQL级别上操作，而是在把结果封送到对象的时候，强制Hibernate过滤掉内存中的重复。很显然，重复在SQL结果中不可避免。

- 基于SQL结果行的查询执行选项（例如利用setMaxResults()/setFirstResult()的分页），如果集合被即时抓取，那么它在语义上就是错误的。如果查询中有一个即时抓取到的集合，在编写本书之时，Hibernate会退回到在内存中限制结果，而不是使用SQL。这样可能会降低效率，因此不建议通过setMaxResults()/setFirstResult()使用JOIN FETCH。如果setMaxResults()/setFirstResult()与JOIN FETCH组合使用，Hibernate的未来版本可能退回到一个不同的SQL查询策略（例如两个查询和子查询抓取）。

Hibernate就是这样实现动态关联抓取的，对于在任何应用程序中实现高性能而言，这是一个基本的强大特性。如13.2.5节中所述，通过查询调优抓取计划和抓取策略是你的第一优化，当越来越多的查询明显会有相同的必要条件时，第二优化便是映射元数据中的全局设置。

最后一个联结选项是theta类型的联结。

5. Theta类型的联结

乘积让你获取两个或者多个类的实例的所有可能的组合。这个查询返回所有有序的User和Category对象对：

```
from User, Category
```

很显然，这通常没用。它常用于一个案例：theta类型的联结。

在传统的SQL中，theta类型的联结是一个笛卡儿积和WHERE子句中的一个联结条件，在乘积上应用它用来限制结果。

在HQL和JPA QL中，当联结条件不是一个被映射到类关联的外键关系时，theta类型的语法就很有用。例如，假设你在日志记录中保存了User的名称，而不是映射一个从LogRecord到User的关联。类与类相互之间不知道对方的任何信息，因为它们没有被关联。然后可以通过下列theta类型的联结，找到所有User和它们的LogRecord：

```
from User user, LogRecord log where user.username = log.username
```

此处的联结条件是username的一个比较，在两个类中都以属性出现。如果两个类都有相同的username，就在结果中被联结（用一个内部联结）。这个查询结果由有序的对组成：

```
Iterator i =
    session.createQuery("from User user, LogRecord log" +
                        " where user.username = log.username")
           .list().iterator ();

while ( i.hasNext() ) {
    Object[] pair = (Object[]) i.next();
    User user = (User) pair[0];
    LogRecord log = (LogRecord) pair[1];
}
```

当然也可以应用SELECT子句，只投影你关注的数据。

可以不需要经常使用theta类型的联结。注意，在HQL或者JPA QL中对两张没有映射关联的表使用外部联结（theta类型的联结是内部联结）目前是不可能的。

最后，执行把主键（或者外键值）与查询参数（或者其他主或外键值）进行比较的查询，这极为常见。

6. 比较标识符

如果以更面向对象的方式考虑标识符比较，实际上就是正在比较对象引用。HQL和JPA QL支持下列查询：

```
from Item i, User u
    where i.seller = u and u.username = 'steve'
```

在这个查询中，i.seller指向ITEM表中USER表的外键（在SELLER_ID列上），并且user指向USER表的主键（在USER_ID列上）。这个查询使用一个theta类型的联结，相当于下面的首选：

```
from Item i join i.seller u
    where u.username = 'steve'
```

另一方面，下列theta类型的联结无法被重新表达为FROM子句联结：

```
from Item i, Bid b
    where i.seller = b.bidder
```

在这个例子中，i.seller和b.bidder是USER表的两个外键。注意这是应用程序中一个重要的查询，用它辨别是谁给他们自己的货品出了价。

你也可能想要把外键值与查询参数进行比较，以便从User中找到所有的Comment：

```
User givenUser = ...
Query q = session.createQuery(
                  "from Comment c where c.fromUser = :user"
                 );
q.setEntity("user", givenUser);
List result = q.list();
```

另一种偶尔会使用的方法是，你喜欢根据标识符值而不是对象引用来表达这类查询。标识符值可能被标识符属性名（如果有）或者特殊的属性名id引用。（注意，只有HQL保证id始终引用任何被任意命名的标识符属性，JPA QL则不保证。）

这些查询相当于之前的查询：

```
from Item i, User u
    where i.seller.id = u.id and u.username = 'steve'

from Item i, Bid b
    where i.seller.id = b.bidder.id
```

然而，现在可能把标识符值用作查询参数：

```
Long userId = ...
Query q = session.createQuery(
                  "from Comment c where c.fromUser.id = :userId"
                  );
q.setLong("userId", userId);
List result = q.list();
```

考虑标识符属性，在下列查询之间有着很大的区别：

```
from Bid b where b.item.id = 1

from Bid b where b.item.description like '%Foo%'
```

第二个查询使用了隐式的表联结，第一个根本没有联结！

这样就结束了我们对涉及联结的查询的讨论。你学习了如何用点号编写简单的隐式内部联结，以及如何用别名在FROM子句中编写显式的内部或者外部联结。我们也探讨了利用外部和内部联结SQL操作的动态抓取。

下一个话题是高级查询，我们认为它们主要用于报表。

14.3.2　报表查询

报表查询利用数据库执行有效的数据分组和聚集的能力。它们天生更为相关；它们并非始终返回实体。例如，不是获取处于持久化状态（和自动进行脏检查）的Item实体，报表查询可能只获取Item名称和初始的拍卖价格。如果这是你的报表屏幕所需的唯一信息(可能甚至是聚集过的、一个类别中的最高初始价格等)，就不需要事务实体实例，并且可以节省在持久化上下文中自动脏检查和高速缓存的过载。

HQL和JPA QL允许使用几个最常用于报表的SQL特性——虽然它们也用于其他目的。在报表查询中，给投影使用SELECT子句，给聚集使用GROUP BY和HAVING子句。

由于已经讨论过基本的SELECT子句，我们将直接跳到聚集和分组。

1. 利用统计函数投影

HQL识别的并在JPA QL中标准化的统计函数是count()、min()、max()、sum()和avg()。

这个查询统计所有的Item：

```
select count(i) from Item i
```

结果作为Long返回：

```
Long count =
    (Long) session.createQuery("select count(i) from Item i")
                  .uniqueResult();
```

查询的下一种变形统计了所有具有successfulBid（排除空值）的Item：

```
select count(i.successfulBid) from Item i
```

这个查询计算所有成功Bid的总数：

```
select sum(i.successfulBid.amount) from Item i
```

查询返回BigDecimal，因为amount属性是BigDecimal类型。SUM()函数也识别BigInteger属性类型，并给所有其他的数字属性类型返回Long。注意SELECT子句中隐式联结的使用：用一个点引用它，来导航从Item到Bid的关联（successfulBid)。

下一个查询给特定的Item返回最低和最高的出价金额：

```
select min(bid.amount), max(bid.amount)
        from Bid bid where bid.item.id = 1
```

结果是一个有序的BigDecimal对（BigDecimal的两个实例，在一个Object[]数组中）。

特殊的COUNT(DISTINCT)函数忽略重复：

```
select count(distinct i.description) from Item i
```

在SELECT子句中调用统计函数时，在GROUP BY子句中没有指定任何分组的情况下，就会把结果压缩到单行，包含统计好的（几个）值。这意味着（缺少GROUP BY子句）任何包含统计函数的SELECT子句都必须只包含统计函数。

对于更高级的统计和报表，需要能够执行分组。

2. 给统计结果分组

就像在SQL中一样，出现在HQL或者JPA QL中但在SELECT子句中的一个统计函数之外的任何属性或者别名，也必须出现在GROUP BY子句中。考虑下一个查询，它统计带有姓（last name）的用户数量：

```
select u.lastname, count(u) from User u
group by u.lastname
```

看看生成的SQL：

```
select u.LAST_NAME, count(u.USER_ID)
from USER u
group by u.LAST_NAME
```

在这个例子中，u.lastname不在统计函数之内，用它给结果分组。也不需要指定想要统计的属性。如果使用一个已经在FROM子句中设置的别名，生成的SQL就会自动利用主键。

下一个查询给每个货品找出平均出价金额：

```
select bid.item.id, avg(bid.amount) from Bid bid
group by bid.item.id
```

这个查询返回有序的Item标识符和平均出价金额值对。注意如何使用id这个特殊属性来指向一个持久化类的标识符，无论标识符的真实属性名称是什么。（同样，这个特殊属性在JPA QL中没有被标准化。）

下一个查询统计出价的数目，并按未售货品计算平均出价：

```
select bid.item.id, count(bid), avg(bid.amount)
from Bid bid
where bid.item.successfulBid is null
group by bid.item.id
```

该查询使用了隐式的关联联结。对于FROM子句中的显式普通联结（不是抓取联结），可以把它重新表达如下：

```
select bidItem.id, count(bid), avg(bid.amount)
from Bid bid
    join bid.item bidItem
where bidItem.successfulBid is null
group by bidItem.id
```

有时候，想要通过只选择一个分组的特定值，来进一步限制结果。

3. 用having限制分组

WHERE子句用来在行上执行限制的相关操作。HAVING子句在分组上执行限制。

例如，下一个查询统计每一个姓中以“A”开头的用户：

```
select user.lastname, count(user)
from User user
group by user.lastname
    having user.lastname like 'A%'
```

管理SELECT和HAVING子句的规则一样：只有被分了组的属性才可能出现在统计函数之外。下一个查询按未售货品统计出价的数目，只给超过10个出价的那些货品返回结果：

```
select item.id, count(bid), avg(bid.amount)
from Item item
    join item.bids bid
where item.successfulBid is null
group by item.id
    having count(bid) > 10
```

大多数报表查询都使用SELECT子句来选择一系列被投影或者统计的属性。你已经见过，当SELECT子句中列出不止一个属性或者别名时，Hibernate就返回查询结果为字节组——查询结果列表的每一行都是Object[]的一个实例。

4. 利用动态的实例化

一般用于报表查询的，字节组（tuple）特别不方便，因此HQL和JPA QL提供了SELECT NEW构造器调用。除了用这种技术动态创建新对象之外，也可以把它与聚集和分组组合起来使用。

如果用含有Long、Long和BigDecimal的构造器，定义一个称作ItemBidSummary的类，就可能用到下列查询：

```
select new ItemBidSummary(
           bid.item.id, count(bid), avg(bid.amount)
           )
from Bid bid
where bid.item.successfulBid is null
group by bid.item.id
```

在这个查询的结果中，每个元素都是ItemBidSummary的一个实例，它是一个Item的概括，包括对这件货品出价的数目以及平均出价金额。注意，此处必须编写一个完全限定的类名，带有包名，除非类已经被导入到HQL命名空间（请见4.3.3节）。这个方法是类型安全的，并且数据迁移类（如ItemBidSummary）可以轻松地对报表中值的特殊格式打印进行扩展。

ItemBidSummary类是一个Java bean，它不一定是被映射的持久化实体类。另一方面，如果通过被映射的实体类使用SELECT NEW技术，查询返回的所有实例便都处于瞬时状态——因此可以利用这个特性填充几个新对象，然后保存它们。

报表查询可能影响应用程序的性能。我们来更深入地探讨这个问题。

5. 利用报表查询提升性能

我们只在Hibernate代码中见过明显的过载（与直接JDBC查询对比）——然后只对非现实的简单测试案例——是在针对本地数据库的只读查询的特殊案例中。在这个例子中，数据库可能在

内存中完全高速缓存查询结果，并迅速响应，因此如果数据集很小，基准一般没用：简单的SQL和JDBC永远是最快的方法。

另一方面，Hibernate甚至通过一个很小的数据集，仍然必须完成把查询结果对象添加到持久化上下文高速缓存（可能二级高速缓存也要）、管理唯一性等工作。如果你曾希望避免管理持久化上下文高速缓存的过载，报表查询就为你提供了一种解决办法。Hibernate报表查询的过载，比起直接的SQL/JDBC通常是不可测量的，即使在非现实的极端案例中，就像从一个本地数据库没有网络延迟地加载一百万个对象一样。

在HQL和JPA QL中使用投影的报表查询，让你指定希望获取哪些属性。对于报表查询，不是选择托管状态下的实体，而只是选择属性或者统计值：

```
select user.lastname, count(user) from User user
group by user.lastname
```

这个查询不返回持久化的实体实例，因此Hibernate不把任何持久化对象添加到持久化上下文高速缓存。这也意味着不必关注处于脏状态的对象。

因此，报表查询导致被分配内存的更快释放，因为对象没有被保存在持久化上下文高速缓存中，直到上下文关闭——执行报表之后，一旦它们没有被应用程序引用，可能就立即被当作垃圾收集起来。

通常情况下，这些考虑极为次要，因此不必去重写所有的只读事务来使用报表查询而不是事务的、被高速缓存的和托管的对象。报表查询更为冗长并且（有争议地）更不面向对象。它们使用Hibernate高速缓存的效率也更为低下，一旦考虑到在产品系统中与数据库进行远程通信的过载，这就更加重要了。你应该等着找到一个在使用这个优化之前就真正有性能问题的案例。

利用目前已经学过的东西，已经可以创建真正复杂的HQL和JPA QL查询了。甚至更高级的查询可能包括嵌套的语句，即众所周知的子查询（subselect）。

14.3.3 利用子查询

SQL一项重要且强大的特性就是子查询。子查询是内嵌在另一个查询中的查询，一般在SELECT、FROM或者WHERE子句中。

HQL和JPA QL支持WHERE中的子查询。FROM子句中的子查询不受HQL和JPA QL的支持（虽然规范把它们列为一项可能的未来扩展），因为这两种语言都没有传递闭包。查询的结果可能不是表格，因此不能在FROM子句中重新用于选择。SELECT子句中的子查询也不受查询语言的支持，但可以通过公式（formula）映射到属性，如8.1.3节中所述。

（Hibernate支持的有些平台不实现SQL子查询。Hibernate只在SQL数据库管理系统提供这个特性时才支持子查询。）

1. 相关的和不相关的嵌套

子查询的结果可能包含单行或者多行。典型地，返回单行的子查询执行聚集。下列子查询返回被一位用户出售的货品的总数量，外部查询返回出售超过10件货品的所有用户：

```
from User u where 10 < (
    select count(i) from u.items i where i.successfulBid is not null
)
```

这是一个相关子查询——它从外部查询引用一个别名(u)。下一个子查询是一个不相关子查询：

```
from Bid bid where bid.amount + 1 >= (
    select max(b.amount) from Bid b
)
```

这个示例中的子查询返回整个系统中最高的出价金额；外部查询返回金额在那个金额一块（美元）之内的所有出价。

注意在这两个案例中，子查询都以圆括号关闭。这通常是必需的。

不相关子查询没有害处，并且方便的时候没理由不用它们，虽然它们通常可能被重写为两个查询（它们不互相引用）。你应该更慎重地考虑相关子查询对性能的影响。在一个成熟的数据库中，一个简单的关联子查询的性能成本类似于一个联结的成本。但是，不一定要用几个单独的查询重写一个相关子查询。

2. 量词

如果一个子查询返回多行，它就是与量词（quantification）组合了。ANSI SQL、HQL和JPA QL定义了下列量词：

- ALL——如果对子查询的结果中所有值的比较为真，表达式就取值为true。如果子查询结果的单个值没有通过比较测试，就取值为false。
- ANY——如果对子查询的结果中有些（任何）值的比较为真，表达式就取值为true。如果子查询结果为空或者没有值满足比较，就取值为false。关键字SOME是ANY的同义词。
- IN——这个二进制的比较操作符可以针对一个子查询的结果比较一系列值，并且如果在结果中找到所有的值，则取值为true。

例如，这个查询返回所有出价都小于100的货品：

```
from Item i where 100 > all ( select b.amount from i.bids b )
```

下一个查询返回所有其他的货品，即出价大于100的：

```
from Item i where 100 <= any ( select b.amount from i.bids b )
```

这个查询返回出价正好为100的货品：

```
from Item i where 100 = some ( select b.amount from i.bids b )
```

这个也一样：

```
from Item i where 100 in ( select b.amount from i.bids b )
```

HQL对在元素或者集合的索引上操作的子查询支持快捷语法。下列查询使用特殊的HQL elements()函数：

```
List result =
    session.createQuery("from Category c" +
                        " where :givenItem in elements(c.items)")
           .setEntity("givenItem", item)
           .list()
```

这个查询返回这件货品所属的所有分类，并相当于下列HQL（和有效的JPA QL），其中子查询更显式：

```
List result =
 session.createQuery(
"from Category c where :givenItem in (select i from c.items i)"
)
.setEntity("item", item)
.list();
```

与elements()一起，HQL还提供indices()、maxelement()、minelement()、maxindex()、minindex()和size()，它们中每一个都相当于针对已被传递的集合的某个相关子查询。参考Hibernate文档，了解更多有关这些特殊函数的信息；它们很少被使用。

子查询是一种高级的技术；你应该质疑子查询的频繁使用，因为带有子查询的查询经常可以只用联结和统计来重写。但是它们有时候很强大并且有用。

14.4 小结

你现在能够在HQL和JPA QL中编写各种各样的查询了。本章介绍了如何准备和执行查询，以及如何绑定参数。我们已经介绍了限制、投影、联结、子查询和许多你可能已经从SQL中知道的其他选项。

表14-4展现了可以用来比较原生的Hibernate特性和Java Persistence的一个概括。

表14-4 第14章中的Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
Hibernate API通过列表、迭代和滚动支持查询执行	Java Persistence标准化了利用列表的查询执行
Hibernate支持具名的和定位的查询绑定参数	Java Persistence标准化了具名的和定位的绑定参数选项
Hibernate查询API支持应用级的查询提示	Java Persistence允许开发人员提供任意的特定于供应商的（Hibernate）查询提示
HQL支持像SQL的限制、投影、联结、子查询和函数调用	JPA QL支持像SQL的限制、投影、联结、子查询和函数调用——HQL的子集

第15章将关注更高级的查询技术，例如利用Criteria API编程式地生成复杂查询，以及原生SQL查询的嵌入。我们也将讨论查询高速缓存，以及什么时候应该启用它。